REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 37	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A Relational/Object-Oriented Database Management System: R/OODBMS (U)

12. PERSONAL AUTHOR(S)

Spear, Ronald L.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 8/90 TO 9/92	1992 September 24	226

16. SUPPLEMENTARY NOTATION

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	database, relational model, object-oriented model, object-oriented programming, heterogeneous database

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

During the last decade, the business sector has become increasingly reliant upon information management. This trend will most likely continue. Deficiencies/constraints in conventional database management systems continue to become more apparent as this reliance continues to grow. Primary areas of deficiency are in modeling, storing, and managing increasingly complex information as in CAD and CASE among others.

The purpose of this thesis is to implement a combined relational/object-oriented database management system that will overcome these deficiencies/constraints. Three possible approaches to such a system exist: build the system from scratch, build object-oriented capabilities on top of an existing relational system, or build relational capabilities on top of an existing object-oriented system. The last approach is the one chosen for this work. This thesis expands previous work in this area and uses a commercial object-oriented database management system, IDB, in its implementation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: *Dr. Michael L. Nelson*	22b. TELEPHONE (Include Area Code): (408) 646-2026	22c. OFFICE SYMBOL: CS/Ne

Approved for public release; distribution is unlimited

A Relational/Object-Oriented Database Management System: R/OODBMS

by

Ronald L. Spear
Captain, United States Army
B. A., Concordia College , 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

ABSTRACT

During the last decade, the business sector has become increasingly reliant upon information management. This trend will most likely continue. Deficiencies/constraints in conventional database management systems continue to become more apparent as this reliance continues to grow. Primary areas of deficiency are in modeling, storing, and managing increasingly complex information as in CAD and CASE among others.

The purpose of this thesis is to implement a combined relational/object-oriented database management system that will overcome these deficiencies/constraints. Three possible approaches to such a system exist: build the system from scratch, build object-oriented capabilities on top of an existing relational system, or build relational capabilities on top of an existing object-oriented system. The last approach is the one chosen for this work. This thesis expands previous work in this area and uses a commercial object-oriented database management system, IDB, in its implementation.

c.1

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank Dr. Nelson, my advisor, for his support and guidance. When I would get bogged down in the coding of our R/OODBMS, Dr. Nelson provided much needed focus which lead to a working system. Throughout the entire thesis process, his thoughtful questions provoked my thinking which helped me to consider more carefully what I was doing and what I needed to do. He could always be counted on to provide timely and constructive feedback.

Thanks also goes to my second reader, Dr. Wu, for his time in effort in assisting with my thesis. He was also quite helpful in the conducting the research necessary for the survey of the literature.

Ellen Borison and John Nestor, of Persistent Data Systems, were of great assistance in making the IDB learning curve a little less steep by providing excellent support. Thank you both for your patience in answering my sometimes trivial questions.

A special thanks goes to my fellow Army officers of class CS11: CPT Walter, CPT (P) Rothlisberger, CPT Nash, CPT Warren, CPT Tharpe, CPT Hoppe, CPT(P) Reese, and CPT Weigeshoff. Without their fellowship, camaraderie, and assistance, I would never have made it to the last quarter with a completed thesis. Their support was invaluable during the first quarter of the curriculum when I had unfortunately missed a third of the quarter.

Finally, I would like to thank my lovely wife Karin whose support has been unwavering throughout our entire time here. Even during my longest days of studying, she never complained and only supported my efforts to succeed here. I could not have done it without her. Bryan and Devin could always be counted on to take my mind off studies when we were together, thank you both.

I. INTRODUCTION

A. MOTIVATION

The popularity of Object-Oriented Programming (OOP) is steadily increasing. As more and more people develop an interest in the additional capabilities that arise from using the object-oriented paradigm, they realize the flexibility and power that OOP provides for representing real world objects in an intuitive and natural manner. This flexibility and high level of abstraction provided by OOP constructs has lead many database professionals to consider object-oriented concepts as they relate to database management systems (DBMS).

During the last decade, the business sector has become increasingly reliant upon information. Information management and control pays dividends to businesses in terms of increased power and revenue. In the coming decade, this reliance will surely increase. The requirement to manage very large quantities of data more efficiently and to perform queries on them rapidly is only part of the problem. The data that businesses wish to store and manage is becoming increasingly complex.

Some specific areas that have an immediate need for a database to store and manage complex data are computer aided design (CAD), computer aided software engineering (CASE), computer-integrated manufacturing (CIM), and computer aided engineering. Data representing images and text is applicable to virtually all large businesses. In the medical field, for example, a patient's medical file could be managed by a database that keeps track of not only the textual records from an appointment, but also x-rays, lab results, cat scan pictures, and other non-textual information. [Me90]

The relational data model has existed for many years. It was developed by Codd in 1970 [Co70]. This model quickly gained widespread acceptance and use commercially.

Much of the business sector has a heavy investment in relational databases, and the relational model is credible and familiar. Therefore, there is a reluctance to change to another model/system, even when there is a need.

Vendors have realized the need for modeling complex information (visual, textual, audible, etc.). In response, several object-oriented databases management systems (OODBMS) are now available in the commercial market. Some businesses with a need for managing complex data are using them. Many of these businesses have expressed a desire for an interface that can access both object-oriented databases and relational databases [St91a].

B. OBJECTIVES OF A RELATIONAL/OBJECT-ORIENTED DATABASE SYSTEM

In general, conventional database models are designed to meet the requirements of a specific need. As Hsiao states, "this is the notion of application specificity, i.e., being specific to a kind of database application."[Hs91, p. 3] The general areas that the conventional data models, relational, hierarchical, network, and functional data models, are used for are record keeping, product assemblies, inventory controls, and inference making, respectively.[Hs91]

The main objective of this research is to determine if a single DBMS can be realized that would serve the needs of both relational and object-oriented users. A relational DBMS (RDBMS) has many advantages over non-relational systems. There are, however, some constraints on relations, such as the type of real world entities that it can represent.

An OODBMS has a more robust capability for representing real world entities as objects. The restriction on types of relations in a RDBMS can be lifted in an OODBMS. Both RDBMSs and OODBMSs have their own set of advantages, but they are not the same set of advantages. It is desirable to have a DBMS that has the advantages found in both of

these sets, plus any additional advantages that may arise from having a combined relational/ object-oriented database management system (R/OODBMS).

The user of a R/OODBMS should be able to implement a relational schema and query the database using either 'standard' relational queries or an object-oriented type query. Thus, there would be a very low learning curve for users already accustomed to relational systems. Additionally, they would not have to abandon the relational approach that they are already familiar with for this new object-oriented paradigm. Yet, they would also gain the capability to manage complex data/objects with the same system.

A secondary objective is to determine if any OODBMS can be the basis for a R/ OODBMS. That is, are there certain requirements that must be met by a commercial OODBMS so that a R/OODBMS can be successfully constructed? Also, are there any characteristics of these systems that would facilitate this construction more than others?

C. RESEARCH OVERVIEW

This is a feasibility study which continues previous work done in the area by Nelson [Ne88] and Filippi [Fi92]. Implementing relational operations in IDB is the primary thrust of this research. Additionally, assessing the capabilities and advantages of using both RDBMSs and OODBMSs falls within the scope of this research. A comparison of these assessments with that of a single R/OODBMS will follow so that it can be determined whether the needs of a RDBMS and an OODBMS can be satisfied by a single R/OODBMS.

II. SURVEY OF THE LITURATURE

A. GENERAL

The focus of this chapter is the fundamental terms and concepts in the areas of object-oriented programming (OOP), relational database management systems (RDBMS), and object-oriented database management systems (OODBMS). The discussion of these topics is not intended to be a complete work on them, but rather an introduction to present those concepts and terms necessary as a foundation for this thesis.

B. OBJECT-ORIENTED PROGRAMMING CONCEPTS

"I have a cat named Trash. In the current political climate, it would seem that if I were trying to sell him (at least to a Computer Scientist), I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented" [Kin89, p. 23].

What does it mean for something to be object-oriented? Since the advent of Simula-67 in the 1960s and later Smalltalk in the 1970s [Mi88][Mo89], object-oriented programming languages (OOPLs) and the object-oriented paradigm have been increasingly great topics for discussion and debate. Different products are advertised as object-oriented, the *hot* buzzword, however there is no universally accepted definition [Kin89][Ne91][Ne90b][SB86].

"What is needed is a definition general enough to encompass all of the current views of OOP, yet strong enough to stand up as the basis for the underlying theory of OOP" [Ne91, p. 4]. The broad definition used in this paper is object-oriented = objects + classes + inheritance + encapsulation. This is a slight modification of the definition in [We87].[1]

Currently, there is also much debate about the design process for an OOP program. In conventional languages, several approaches have been formalized such as the top-down, bottom-up, structured, etc. These approaches are, in general, 'action' approaches while OOP uses an 'object' approach [GH91b]. However, the OOP community lacks any established methodology to their 'object' approach [PN91b]. This subject is considered to be beyond the scope of this paper, however, and as such will not be discussed any further.

1. Classes/Objects

A *class* may be defined as "a description of one or more similar objects" [SB86, p. 43]. An *object* is an instantiation of a class [Mo89]. Clearly, this is a circular definition which can be avoided by defining an object as the fundamental element of OOP. That is, an *object* is a self-contained set of variables (which may be thought of as attributes in database terms), and responds only to messages to execute specific defined procedures (also called *methods* in OOP terms) [BM91][Ne91][SB86]. An object's method(s) are the only means by which manipulation of its variables can occur. [2]

The description of an object/class is composed of variables/attributes and the procedures/methods that operate on them. These variables and methods describe general characteristics that all instances of a class have. The only way to communicate with an object is through messages to execute its methods. Thus, an object's messages are the interface to a particular object. An important point is that the variables of an object may

---

1. [We87] defines OOP as "object-oriented = objects + classes + inheritance". It may be argued that this equation is the same as our slightly modified equation since some definitions of class imply encapsulation. For a more detailed discussion, see [Ne91]

2. Thus, a class may be thought of as an abstract data type (ADT) or as the implementation of an ADT [Da84][Ne91]. The interface to the data type is defined solely by the methods defined for the class. The implementation of these methods is contained within the class which allows the interface to be implementation-independent.

themselves be objects. In this case, the object is called a *composite object* that has at least one variable which is a previously defined object [EN89][Ne90b]. The composition of composite objects can be compared with inheritance where composition is a form of part inheritance the later is concerned more with behavior inheritance [Ne90b].

Most OOP languages provide for both class and instance variables, although it is not required for a language to be considered object-oriented [Ne91][Ne90b]. The difference is that a *class variable* will have the same value for all instances of the class. If a method in any of the instances of that class change the value of a class variable, then the value is changed for every instance of the class. Thus, it is a variable shared by all instances of the class. In contrast, an *instance variable* has a local value for a particular instance. A change to the value of an instances instance variable of one object has no effect on the corresponding variable in another instance of the class.[3]

Consider a military officer as an example of an entity that might be described in a class definition. The class name could be Officer with several variables that further characterize an officer. A possible class variable could be the number of company grade officers (O-1 through O-3) that exist: CompanyGradeCount. Thus, every instance of the Officer class would have the same value for CompanyGradeCount. Each officer has a name, grade, social security number, and an assigned unit that may be represented by the instance variables Name, Grade, SSN, and Unit, respectively. These variables would be instance variables since each instance of an officer will not necessarily have the same values for each of these variables. This officer class definition is presented in Figure 1.

---

3. IDB, the OODBMS used in this thesis, does not provide the capability to define class variables. However, in our discussions with Persistent Data Systems, they have mentioned that later versions of IDB may contain this capability.

Class: Officer	
Superclass:	**none**
Class Variables:	**CompanyGradeCount**
Instance Variables:	**Name, Grade, SSN, Unit**
Methods:	**Promote, Retire, Relocate**

Figure 1 Class Definition Example

2. Methods

As previously mentioned, *methods* are the means by which an objects variables are manipulated [Ne90b]. They define the only legal actions/operations that characterize an object. A method is invoked by sending an object a message to invoke one of its methods. A message can be likened to a procedure call in a conventional programming language.[4] Changing the implementation details for any method should have no effect on messages needed to invoke that method [Ne90b]. Thus, code that uses message passing for method invocation is implementation independent. It is in this sense that the description/definition of class may imply encapsulation (also called information hiding).

In our military officer example, it may be desirable to be able to promote, retire, or relocate an officer. Therefore, these three actions might be implemented as methods for the Officer class (see Figure 1). The Promote method would change the Grade instance variable and could change the CompanyGradeCount class variable; Relocate would change

4. The exact implementation for the message passing concept may vary among different OOP languages.Value operations (idl_vop) and type operations (idl_top) are the means by which IDB implements message passing.

the Unit instance variable; and Retire would change the Unit and Grade instance variables, and possibly the CompanyGradeCount class variable.

3. Inheritance

Inheritance allows further specialization of a class (called the superclass) by exploiting class similarities [Ni89]. That is, a subclass inherits the variables and methods of its ancestors and adds its own variables and methods.[5] In this way, it is a specialization of its ancestors and its ancestors are a generalization of the subclass [BM91][CY90][HO87][Hs91][Mi88][Mo89]. In more fundamental terms, it is a form of code sharing [Ne90b] that facilitates and encourages code reuse [Ni89][Mi88]. As you travel down an *inheritance hierarchy*[6], classes become more specialized. As you travel up, they become more generalized.

In Figure 2, the class Human is the superclass of the subclass ServiceMember. A simple inheritance hierarchy of these two classes is shown in Figure 3. Any ServiceMember will have Species as a class variable; Weight, Height, Sex, and Ssn as instance variables; and the methods Born, Die, EnterService, and DepartService. A ServiceMember is a specialization of a Human, while a Human is a generalization of a ServiceMember.

---

5. Inheritance of methods is sometimes called behavioral inheritance while inheritance of variables is called structural inheritance [Da90].

6. An inheritance hierarchy is simply a graphical representation of the inheritance relationship between classes [Ne90b].

Class:	**Human**
Superclass:	**none**
Class Variables:	**Species**
Instance Variables:	**Weight, Height, Sex**
Methods:	**Born, Die**
Class:	**ServiceMember**
Superclass:	**Human**
Class Variables:	**none**
Instance Variables:	**Ssn**
Methods:	**EnterService, DepartService**

Figure 2 A Class and Its Subclass

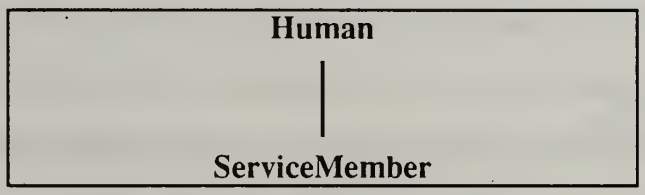


Figure 3 A Simple Inheritance Hierarchy

Consider the inheritance hierarchy in Figure 4, Officer[7] is now a subclass of ServiceMember. The ancestors of Officer are Human and ServiceMember, while its descendents are the subclasses CompanyGrade, FieldGrade, and FlagGrade. Thus, the class Officer inherits all the variables and methods from both Human and ServiceMember. Again, you can see that as you travel down the inheritance hierarchy the classes become more specialized while the classes higher in the hierarchy are more general.

7. The Officer class definition in Figure 1 would have to be modified to reflect that ServiceMember is now its superclass.

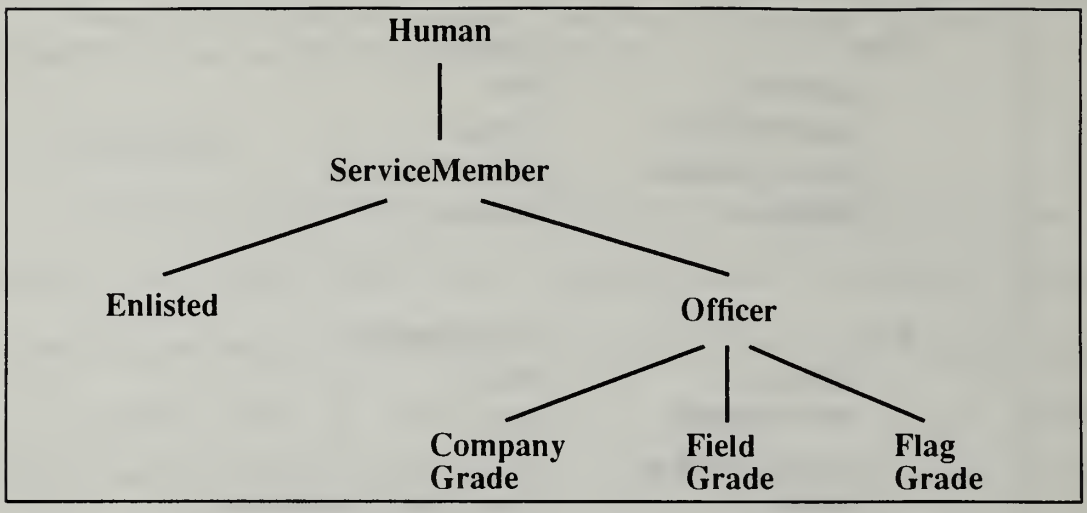


Figure 4 A More Complex Inheritance Hierarchy

Methods that are inherited by a class may have their implementation overridden (or redefined) while still maintaining the same method name [HO87][Mi88]. This allows methods to be overloaded (also called *polymorphism*) [Ne90b][Ni89][SB86]. That is, the same message may be sent to different objects, invoking different implementations of the same method depending on the object which receives the message. A common example of polymorphism is that of the binary arithmetic operations on integers and real numbers. When we want to add two integers, we write 2+39; the same operator name '+' is also used for the real numbers, 3.4+2.1.

In the example hierarchy (Figure 4), the subclasses of Officer all inherit the method Promote (along with the other methods of the Officer class). However, each of the subclasses (except for CompanyGrade) require a different implementation for the Promote method since promoting a FlagGrade or FieldGrade officer would not change the Officer class variable CompanyGradeCount.[8] A method that has been overwritten by a subclass is a polymorphic method.

Up to this point, our discussion of inheritance has focused only on *single inheritance*: a class inherits from only one superclass.[9] However, some languages allow for *multiple inheritance*: a class inherits from two or more superclasses [Ne91][Ne90b]. The relationship between classes when multiple inheritance is allowed can be shown graphically in a multiple inheritance *lattice*[10] (see Figure 5) [SB86][Ne90b].

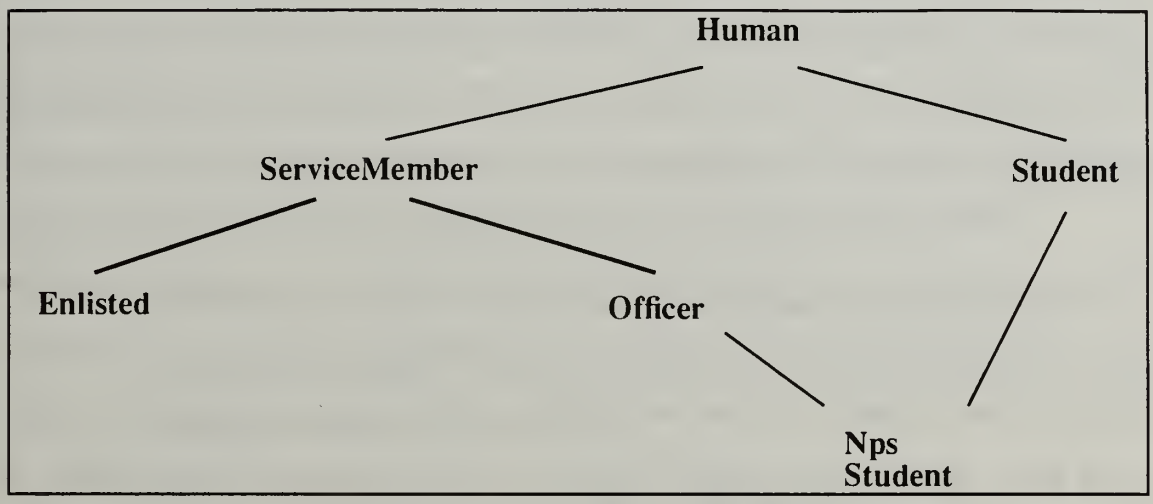


Figure 5 A Multiple Inheritance Lattice

Inheritance is not without its own special problem which must be mentioned; that of name conflicts [Ki91][Mi88][Ne88][NMO90]. In Figure 5, assume that the Human class

8. This assumes that in the Officer class definition the Promote method has the implementation previously described: Promote changes the value of the CompanyGradeCount class variable. Thus, if the method's implementation is not overwritten a message sent to any of Officer's subclasses would invoke this implementation causing undesirable results.

9. Generally, the term inheritance by itself means only single inheritance [Ne90b].

10. The term lattice by definition allows more than one superclass [SB86] while an inheritance hierarchy does not. This is in contrast to a hierarchy which is represented as a tree. However, it is fairly common practice to refer to a multiple inheritance lattice more simple as a multiple inheritance hierarchy [Ne90b].

has a Print method that has been overridden in both the Officer and Student subclasses. Now, consider the class NpsStudent; which implementation of Print does it inherit?

In the case of single inheritance, a subclass may inherit a method X or variable Y from its ancestors but also have a different method X or variable Y defined locally for the subclass [Ni89][NMO90]. That is, the subclass also has in its class definition a method X and a variable Y. How are the two different methods or variables differentiated? Usually locally defined methods and variables override any inherited methods and variables with the same name. However, some systems allow the overridden method/variable in the superclass to be accessed by prefacing the name with super; for example, superX or superY.

Multiple inheritance has a similar name problem, however it is complicated since there is more than one superclass involved. It is not as simple to just preface a redefined name with super since more than one of the superclasses may have the same method/ variable name. In general, the possible solutions include making a choice between them using some default criteria [SB86] (such as a class precedence [Mo89][SB86]), by distinguishing among them [Ne88] (possibly by using their parent class name as a prefix [Ni89]), by combining them [Mo89][Ne88], or by requiring the programmer to make an explicit choice [Ki91][Mi88][Ni89].

4. Encapsulation

"An OOPL supports encapsulation if it allows users of objects to access them only via their external interfaces" [Mi88, p. 15]. The external (public) interfaces to an object are its methods and these are the only means that a user has for accessing/manipulating an object. The user is not allowed direct access to the object and its inner workings[Da90]. Thus, *encapsulation* is a method of information hiding [Da90][Ne90b][Ne91] and offers protection to objects from unauthorized/illegal operations on its variables [Mo89]. Additionally, hiding implementation details provides a separation (or decoupling) from the

code that defines and implements an object from that of the program code using the object [Da90]. This point is quite important since it allows changes to an object's implementation without having to change the user's program code and vice-verse [Da90][Mo89][Ne91].

C. RELATIONAL DATABASE MANAGEMENT SYSTEMS

Prior to discussing the relational model specifically, it is worth giving a few moments to the definition of a database management system (DBMS). A DBMS is a collection of general-purpose software programs that allow maintenance of and access to a collection of interrelated data [EN89][KS86]. A collection of interrelated data is called a *database.*

However, when discussed in terms of specific DBMSs, the definition of database is a little more restrictive and has the following properties[EN89]:

- the collection of data has some inherit meaning and is logically coherent;
- the database is created for a specific purpose, an intended group of users, and some preconceived applications; and
- the database models and reflects some real world aspect.

The maintenance of and access to the data include facilities to define, create, and manipulate (i.e., query and update) it.

There are numerous considerations for using a DBMS. Controlling redundancy: as the number of times the same data is stored increases, there is a corresponding increase in duplication of effort to update it; additionally, it is more likely that inconsistencies among the data may arise (update anomalies). Sharing of data: in a multiuser DBMS concurrency control of updates is critical to the correctness of updates. Restricting unauthorized access: the DBMS should allow restrictions to be placed on access to database data. Representing complex relationships among data: this includes easy and efficient retrieval and update of related data. Enforcing integrity constraints: database designers must be allowed to specify data constraints which the DBMS can enforce automatically or which can be enforced by

update programs. Provide backup and recovery: recovery from hardware and software failures is the DBMS's responsibility. [EN89][KS86]

1. Relational Model Concepts

In 1968, Dr. Edgar F. Codd had the idea that "predicate logic could be applied to maintaining the logical integrity of the data" in a DBMS [CD90, p. 35]. This was the conception of the relational data model. Two years later, Codd introduced his model in a paper published in the Communications of the ACM [Co70]. It was a departure from what had until that time been the conventional models (hierarchical data model and the network data model) since his model allowed for a more abstract representation of the database [OV91][KS86][Fi92]. This simplistic yet complete model has evolved into a kind of defacto standard in the database industry.

The relational model is firmly founded in strong mathematical concepts and theory. Predicate logic and set theory are the primary foundation upon which the relational model rests. This allows a formalism in the way that data is represented and manipulated in the context of the model.

a. Relations

The relational model represents a database as a collection of *relations*. Relations, the fundamental building block of the model, are represented by the intuitive notion of a *table* (or flat file) of values [Da84][EN89][KS86][OV91][SSU91]. Each row of the table represents a tuple of the relation. A *tuple* is a collection of data values that are related. The columns of the table represent *attributes* of the relation (see Figure 6). In the sample relational database, there are two relations: Officer and Military Unit. Each tuple is interpreted as a fact that describes a relation instance. Every attribute value within a tuple must be atomic. That is, it is not constructed of other components; it is indivisible.[11]

Officer

Name	SSN	Unit
Spear	550-34-2453	3BDE
Walter	233-45-3423	2BDE
Nash	241-4500974	1BDE
Rothlisberger	123-45-6789	3BDE

MilitaryUnit

UnitName	Location	CdrSSN
1BDE	Grafenwohr	123-45-6789
2BDE	Erlangen	550-34-2453
3BDE	Bamberg	233-45-3423

Figure 6 Sample Relational Database

However, it must be understood that there is a subtle but important difference between a relation and a table. A relation is a set of tuples. On the other hand, tuples of a relation are represented as the rows of a table. A set by definition is unordered, but clearly the rows of a table are ordered (from top to bottom)[EN89]. Thus, the relation Officer in Figure 7 is the same as the Officer relation in Figure 6.

11. This requirement for atomic attributes is called the first normal form. Normalization is a process of decomposing relational schemas into smaller relations that conform to several criteria: first, second, and third normal forms. The goal of normalization is to aid database designers in analyzing a database schema and developing a database with a 'good' design that avoids update anomalies. [EN89]

By the same token, the order of the columns is not important since a tuple can be thought of as a set of (<attribute>,<value>) pairs[EN89][Da84]. The value in each pair must fall within the domain of its associated attribute. Thus, the first tuple in Figure 7 could be written: (Name,Rothlisberger), (SSN, 123-45-6789), (Unit,3BDE); or (Unit,3BDE), (Name,Rothlisberger), (SSN, 123-45-6789); etc.

Officer

Name	SSN	Unit
Rothlisberger	123-45-6789	3BDE
Nash	241-4500974	1BDE
Spear	550-34-2453	3BDE
Walter	233-45-3423	2BDE

Figure 7 Reordered Officer Relation

The reason this subtle difference between relations and tables is important will become more apparent later. As will be discussed, all of the relational algebra operators take relations as parameters and yield results that are also relations [Da84]. Thus, they are set operations and they have firm foundations in mathematical set theory. Consequently, if someone has a good understanding of mathematical set theory, then they are well on their way to understanding relational algebra.

Another consequence of a relation being defined as a set of tuples, is that, by definition, each element in a set is unique. Therefore, all tuples in a relation must be unique. This implies that there are no two tuples in a relation that have the same combination of values for all attributes [EN89]. Thus, there is some combination of attributes that allows each tuple of a relation to be uniquely identified; this combination is called a *primary key* (the primary key of each relation in Figure 6 and Figure 7 is underlined). In the worst case,

the primary key is the set of all attributes of the relation [Da84][EN89]. A detailed discussion of keys (including superkeys, minimal superkeys, candidate keys, primary keys, and foreign keys) can be found in [EN89].

b. Schemas and Constraints

When discussing databases, it is important to differentiate between the database *schema* and database *instance*. A database *schema* is a set of relation schemas and a set of integrity constraints while an *instance* is a snap-shot of the data in the database at a given instant in time[KS86][EN89]. A relational schema can be thought of as a template for the relation. The type definition in programming language notation bears a close correspondence to a relation schema [KS86]. The schema is known, in database terms, as *intention* of the relation while the instance is the *extension* [EN89].

In the previous examples (Figure 6 and Figure 7), the Officer relation and the Military Unit relation are both examples of a relation instance. Figure 8 shows the relational schemas for the Officer and Military Unit relations. Another notation for a schema is a listing of the relation's attributes and their corresponding domains[KS86]:

- Officer-scheme = (Name : string, SSN : integer, Unit : string)
- Military Unit-scheme = (UnitName : string, Location : string, CdrSsn : integer)

Officer

Name	SSN	Unit

MilitaryUnit

UnitName	Location	CdrSSN

Figure 8 Sample Relational Database Schema

In general, *integrity constraints* are defined for a database schema and should hold for every instance of the schema [EN89]. There are several types of integrity constraints: key constraints, entity integrity constraints, and referential integrity constraints [Da84][EN89][OV91]. *Key constraints* are concerned with ensuring the uniqueness of key values for every tuple in any relation instance of a schema. *Entity integrity constraints* restrict primary key values to be non-null [OV91]. Finally, *referential integrity constraints* maintain consistency between tuples of two relations [OV91]. That is, if a tuple in one relation refers to another relation, then it must refer to a tuple that exists in that relation. An additional general constraint that is sometimes needed is called a *semantic integrity constraint* [EN89]. In our military example, a semantic integrity constraint would be that 'an officer in a unit cannot outrank the commander of his unit'.

c. Operations

Three fundamental update operations exist for relations: modify, delete, and insert. Delete and insert do what is expected - insert and delete tuples, respectively. Modify allows values of some or all attributes of a tuple to be changed. The most important aspect in performing these operations is to ensure that the integrity constraints defined for the schema are not violated. Therefore, if any integrity constraints would be violated by the update operation, then either the operation can be rejected or the system may attempt to correct the reason that the operation violates integrity constraints. [EN89]

It is possible for an insert operation to violate all three of the integrity constraints: key constraints, entity integrity constraints, and referential integrity constraints. However, the delete operation can only violate the referential integrity constraint. Both key constraints and referential integrity constraints can possibly be violated by a modify operation. [EN89]

2. Formal Query Languages

Along with Codd's initial presentation of the relational model in 1970, he also proposed two formal query languages: relational algebra and relational calculus [Co70]. Before discussing the fundamental differences in the two, it should be noted that it has been shown that they are equivalent in expressive power [EN89][OV91]. Thus, any query that can be specified in one language can also be specified in the other. Query languages can therefore be compared against either of these two to determine if they are relationally complete [EN89].

Relational algebra, in contrast to relational calculus, is more closely related to the underlying system instructions that perform the operations [KS86]. This is the primary reason that relational algebra was chosen over relational calculus for use in this thesis. Since the two are logically equivalent, it is sufficient to implement only one. Thus, in the discussion that follows, relational algebra will be discussed in more detail than relational calculus.

The fundamental difference between the two languages is that of a procedural language verses a nonprocedural (or declarative) language. In a procedural language, such as relational algebra, the query specifies the sequences of instructions necessary to obtain the result. This is in contrast to nonprocedural language, such as relational calculus, which specifies a query by delineating what information is desired rather than how it is to obtain the information (i.e., the procedure to be followed). [EN89][KS86]

a. Relational Algebra

Relational algebra is a procedural language. Therefore, it more directly corresponds to the operations necessary to satisfy a query. The set of operations that constitute relational algebra are derived from the mathematical theory of sets [OV91]. Each operation has as its parameter(s) a relation and returns a result that is also a relation [Da84].

Codd proposed eight operations in his presentation of the relational model in 1970: union, intersection, difference, Cartesian product, select, project, join, and divide [Da84]. However, it can be shown that of the eight only five are primitive operations: select, project, union, difference, and Cartesian product [Da84][EN89][KS86][NMO90]. The other three non-primitive operations can be formed by some combination of the five primitive ones.

(1) Select is a unary operation, taking a relation as its parameter and yielding a subset of tuples from that relation as its result. The resultant relation has the same relational schema as the input parameter relation. To identify the tuples that are to be in the resultant relation, a selection condition (predicate) is specified in the select expression on the specified relation. All tuples in the resultant relation will satisfy the selection condition. [EN89][KS86]

The selection condition is a boolean expression consisting of clauses of the form:

<name of attribute><comparison operator><constant value>;

or

<name of attribute><comparison operator><name of attribute>

where <name of attribute> is the name of an attribute of the input parameter relation; <comparison operator> is either =, <, ≤, ≥, >, or ≠; and <constant value> is any arbitrary number of clauses may be connected with the AND, OR, and NOT operators to form the selection condition expression [EN89].

In Figure 9, ResultRelation is the resultant relation from performing a select operation on the relation Officer (Figure 7) where Unit = 3BDE. The original Officer relation had 4 tuples while the resultant ResultRelation has only 2. The selection condition can be arbitrarily complex.

ResultRelation

Name	SSN	Unit
Rothlisberger	123-45-6789	3BDE
Spear	550-34-2453	3BDE

Figure 9 Example Result of a Select Operation

(2) Project is also a unary operation. The project operation uses an attribute list to select the attributes that will appear in the resultant relation. The attribute list must be a subset of the attributes of the input parameter relation. The schema of the resultant relation in this case is not the same as the input parameter relation. The attributes are a subset of the attributes in the input parameter relation. However, the attributes in the result will be listed in the order that the attributes are listed in the attribute list. However, the resultant relation will have the same number of tuples. [EN89]

In Figure 10, ResultRelation is the resultant relation from a projection operation of the Name and SSN fields of the Officer relation (Figure 7). The original Officer relation has three attribute fields while the resultant ResultRelation only has two. However, both relations have the same number of tuples.

ResultRelation

Name	SSN
Rothlisberger	123-45-6789
Nash	241-4500974
Spear	550-34-2453
Walter	233-45-3423

Figure 10 Example Result of a Project Operation

(3) Union, in contrast to the first two operations, is a binary operation. It along with the operations difference and Cartesian product are the standard mathematical set operations [Da84][EN89]. Thus, the union of two relations is a resultant relation that is the set of all tuples that belong to either relation or to both. This operation, along with the difference operation, can only be executed if the two input parameter relations are union compatible. That is, they must have the same number of attributes (the same degree) and the corresponding attribute must be based on the same domain (although they need not have the same name) [Da84]. As an example, two union compatible relations are shown in Figure 11. The resultant relation has the same degree as the input relations. Note that the order of the attributes in the relations is important for checking union compatibility.

OfficerReserve

FirstName	LastName
Ron	Spear
James	Schledorn
Leonard	Tharpe

EnlistedActive

Fname	Lname
James	Justice
John	Walter
Luther	Moen
Jane	Pauli
James	Schledorn

ResultRelation

first_name	last_name
Ron	Spear
James	Schledorn
Leonard	Tharpe
James	Justice
John	Walter
Luther	Moen
Jane	Pauli

Figure 11 Union Compatible Relations and Result of Union

(4) Difference, given two input parameter relations, A and B, produces a resultant relation which is comprised of all tuples in A that are not in B. As with the union operation, the input parameter relations A and B must be union compatible. Therefore, the difference of relations OfficerReserve and EnlistedActive (from Figure 11) is shown in Figure 12. However, the difference of EnlistedActive and OfficerReserve would be a resultant relation that contains all of the tuples in relation EnlistedActive except for the last tuple which contains James Schledorn and no tuple from OfficerReserve.

OfficerEnlisted

first-name	last-name
Ron	Spear
Leonard	Tharpe

Figure 12 Result of Difference Operation on Relations in Figure 11

(5) Cartesian Product, the last of the set operations, differs from the previous two operations, union and difference, in two main ways. First, if the Cartesian product operator is operating on relations A and B which have 2 tuples with 3 attributes and 5 tuples with 4 attributes respectively, then the resultant relation will have 10 (2 * 5) tuples and 7 (3 + 4) attributes. In our previous example (Figure 11), the Cartesian product of the two relations would have a resultant relation with 15 (3 *5) tuples and 4 (2 + 2) attributes. Secondly, the two input parameters to this operation need not be union compatible.

b. Relational Calculus

While relational algebra is founded on the mathematical principles of set theory, relational calculus (a procedural language) is founded on first order predicate calculus [EN89]. Two well-know forms comprise the relational calculus: tuple relational

calculus and domain relational calculus. There is a strong similarity between the two forms. As their names imply, *tuple relational calculus* is the form in which variables represent tuples while *domain relational calculus* uses variables to represent attribute domain values [KS86]. In tuple relational calculus, the database is viewed as a set of tuples, while in domain relational calculus it is viewed as a set of domains. Thus, meaning is given to queries by interpreting variables as assertions on the database [OV91].

3. Other Query Languages

SQL, QUEL, and QBE are three commercial query languages that expand on the formal languages previously discussed, relational algebra and relational calculus. Both SQL and QUEL are relationally complete languages, however, some implementations of QBE are not (they lack explicit universal and existential quantifiers)[EN89]. In general, they provide a higher level, more friendly user interface along with other facilities for data definition (a data definition language function, DDL), data manipulation (data manipulation language facilities, DML), and security constraint specification[12] [KS86]. It is considered beyond the scope of this thesis to discuss these other areas. However, each of these languages will be reviewed very briefly in this section to give the reader a feel for considerations in implementing a commercial query language. All of these languages are more declarative than procedural.

a. SQL

The Structured Query Language, SQL, is still referred to as Sequel (Structured English QUEry Language) by many people and is probably the most well known of the three. SQL combines constructs from both relational algebra and calculus

12. Although, each language does not necessarily provide for all of these facilities.

[EN89][KS86]. It has become somewhat of a defacto standard relational database language. Queries in SQL can be interactive or embedded in an application [Da84].

The basic format for a SQL query is:

SELECT <list of attributes>

FROM <list of tables>

WHERE <condition>

where <list of attributes> is a list of names for which values are desired (corresponds to the relational algebra project operation), <list of tables> is a list of required relations necessary to satisfy the query, and <condition> identifies the tuples desired by evaluation of a boolean expression (corresponds to the relational algebra selection condition in a select operation [EN89].

One departure from the formal relational models is that SQL allows tuples within a relation to be repeated. Thus, a relation in SQL is not a set of tuples but a multiset (or bag) [EN89].

Some current work in the area of query languages is being done using the idea of a graphical interface for constructing queries. One such effort attempts to overcome the difficulties in forming SQL queries (ease-of-use) by designing and implementing a relationally complete graphical dataflow query language (DFQL) [Cl91].

b. QUEL

QUEL has similar functionality to SQL; however, instead of combining constructs from both of the formal languages, it closely parallels tuple relational calculus [KS86]. Three types of clauses are generally used to construct most QUEL queries: range of, retrieve, and where [KS86]. *Range of* explicitly declares the tuple variables, *retrieve* declares the attribute to retrieve (corresponds to projection attribute list), and *where* specifies the selection condition [EN89].

c. QBE

Query by Example, QBE, is most closely related to domain relational calculus. It is unique in that it is a two dimensional language: skeleton tables (displaying the relation schema) are displayed pictorially, and queries are then expressed by filling in an *example* row(s). When looking at a QBE query, constant values appear without any special markings or indicators while domain variables (which do not have to match any specific database values and are completely arbitrary) are indicated by being preceded by an underscore ('_'). To specify that the values of a certain column are to be retrieved, the prefix 'P.' (for print) is used. [EN89][KS86]

D. OBJECT-ORIENTED DATABASES

As with object-oriented programming, the relatively new area [Ne90a] of object-oriented databases (OODBMS) does not as yet have a formal definition/specification or even an agreed upon informal definition/specification [Ed91][US90]. "Object-oriented databases = object orientation + database capabilities" [Kh91, p. 31] is one attempt to define OODBMSs where the database capabilities alluded to include persistence, transactions, concurrency control, recovery, querying, versioning, integrity, security, and performance issues [Kh91]. This could be rewritten as object-oriented databases = object-oriented + database capabilities[13].

This section provides a general description of OODBMS concepts along with a discussion on several OODBMSs currently available: IDB, Ontos, Vbase, Gemstone, and POSTGRES. Of the systems looked at, only IDB will be given more than a cursory look

13. [Kh91] defines object orientation as abstract data types + inheritance + object identity, which is essentially the same definition previously used in this paper for object-oriented. For a more in-depth discussion of OODBMS concepts see [US90].

since it is the system used to implement the Relational/Object-Oriented Database Management System in this thesis.

1. Object-Oriented Model Concepts

The most well known data modeling model is the Entity-Relationship (ER) data model. Until the late 1970s, this model was sufficient for supporting the modeling needs of conventional DBMSs (hierarchical, network, and relational systems) that meet traditional business data processing requirements [EN89]. The ER data model is a high-level model used prior to actually developing a database schema in a specific DBMS. As modeling needs have become more complex, the ER data model has become increasingly inadequate. To meet the needs for complex data models, many data models have been proposed. Of those proposed, they generally fall into three categories: semantic data models[14], functional data models, and object-oriented models [EN89].

In conventional DBMSs, data is modeled using the classical record-oriented ER model. Here data is looked at as a group of relations (or record types) that are comprised of a group of tuples (or records) which are all stored in a file [EN89]. Thus, the ER model must be converted into a DBMS specific model (or schema) which may lose its resemblance to any real world entities (or objects) during the conversion process. A good example is that of an ER model for a relational database application which is converted to a group of relations that are then normalized. In the normalization process, the original model may be distorted to such an extent that any relationship to the original real world entities being represented in the database application is lost as information is scattered among relations [EN89].

---

14. [EN89] provides an detailed discussion of what they term the enhanced-ER model (EER) which encompasses what they consider the most important concepts of the semantic data model.

Of the models mentioned above, we focus on the object-oriented data model. With an object-oriented model, a database is considered group of objects that represent real world entities [EN89]. In OODBMSs, objects are represented directly by database objects [EN89]. Thus, there is no loss of the original model as with the ER model; as more complex real-world entities are modeled using complex objects, there is a direct correspondence between a real-world entity and its database representation. This direct correspondence allows objects to maintain their integrity and identity which in turn allows the objects to be identified and operated upon [EN89]. Some have gone so far as to question whether it is meaningful to talk about an object-oriented data model [Ki90] since object-oriented data model concepts and the object-oriented paradigm are for all practical purposes the same.

Fundamental object-oriented data model concepts include data abstraction, encapsulation, object identity [Mc91], inheritance, complex objects, message passing, and operator overloading (or polymorphism) [Ki90]. All of these concepts, with the exception of object identity, were specifically addressed previously in this chapter under object-oriented programming concepts. Generally, each object is represented by an object identifier that is system generated. The identifier is independent from any key attributes which allows attributes to be modified without destroying the objects identity. [EN89]

2. Object-Oriented Database Systems

Persistence of objects is considered the primary difference between OOPLs and OODBMSs [EN89]. Objects used in an OOPL program exist only during program execution whereas those in OODBMSs must exist permanently in secondary storage from session to session. Thus, the OODBMSs and OOPLs are quite similar except for additional facilities provided by the OODBMS system. By the same token, OODBMSs have similar advantages as those of OOPLs: expressibility, reusability, etc.

"The Object-Oriented Database System Manifesto"[15] written in 1989 by M. Atkinson, F. Bancilhon, D. DeWitt, K. Kittrick, D. Maier, and S. Zdonik for the First International Conference on Deductive and Object-oriented Databases, Kyoto, Japan, describes 13 mandatory characteristics, listed in Figure 13, for a database system to be considered an OODBMS [Ed91].

1. supports complex objects

2. supports object identity

3. encapsulates objects

4. supports either types or classes

5. classes or types inherit from their ancestors

6. do not bind prematurely

7. are computationally complete

8. are extensible

9. data is persistent

10. manages very large databases

11. allow concurrent users

12. recover from software and hardware failures

13. have a simple way to query data

Figure 13 OODBMS Manifesto

15. This does not contain an agreed upon set of characteristics, rather, it is an attempt to offer characteristics for agreement within the database community.

There are some correlations that may be made between RDBMS and OODBMSs that help to conceptualize how some of the object-oriented paradigm relates to the relational paradigm. A row in a relation may be thought of as an object in an OODBMS. The set of rows of a relation may be equated to a class. Other concepts in OODBMSs that have no correlation to anything in a relational system include: methods, object identifiers, inheritance, and encapsulation. However, OODBMSs do not have a mathematical foundation to stand on as relational systems do. [Ed91]

3. IDB Object Database Overview

a. General Information

IDB [Pe91a][Pe91b][Pe91c][Pe91d] is a new OODBMS that first entered the commercial market in 1990 . However, it could be considered to have been in development for over a decade with the design and architecture of the Interface Description Language (IDL), a subset of which is a fundamental component of the IDB system. IDB was built from the ground up as an object database management system. This is in contrast to some other database management systems that claim to be object-oriented but are not 'real' object-oriented systems. Some of these are systems that have as their kernel or core a relational system that has object-oriented extensions added on top. Others claim to be object-oriented but fall short in fully implementing the concepts that comprise the object-oriented paradigm: inheritance, encapsulation, polymorphism, and abstract data types.

The current version of IDB is 1.1. Version 1.0 ran on several platforms: Domain OS (680X0), Sun 3 (680X0), Sun SPARC and Windows 3.x. Version 1.1 extends this list to include the Macintosh, NeXT, and HP-UX (680X0 and PA-RISC). Other key differences between the version 1.0 and 1.1 include: supported platforms that are networked together can now share data over the network; attributes may be removed from a schema without having to modify the ASCII form file; ease of access to menu facilities

has increased; development tools may be used with Windows 3; and bugs have been removed that existed in version 1.0. [Pe91b]

Persistent Data Systems, the designer and developer of IDB, describe IDB as "an object database for software developers who build applications that must manage complex shared data"[Pe91d, p. 3]. Applications suited for use with IDB include CASE, CAD, image management systems, hypertext systems, hypermedia systems and geographic information systems. This list is not intended to be all inclusive, but rather to give the reader an idea of the types of applications with requirements to model, manage, and store complex and unconventional data. [Pe91d]

The data definition language (or schema language), IDL, extends Kernighan and Ritchie (K&R) C [KR78] to include object-oriented capabilities. Polymorphism, multiple inheritance, and dynamic binding and loading are among these capabilities. Because IDB uses K&R C, tools available for working with C (such as the C compiler and on-line debugger) on each supported platform may be used in the development of IDB applications. [Pe91d]

IDL facilitates the interface between C and IDB. The interface can be thought of as consisting of three parts: core interface, display manager interface, and browser interface. The core interface allows IDB applications to have object-oriented capabilities. Display management for stand alone applications uses the display manager interface. This interface is not exclusive to stand alone applications since its features can also be accessed from within the browser. The browser interface has the features of the display manager interface in addition to other special features only available in the browser. [Pe91d]

b. Clusters and Structures

A key concept in IDB is that of a 'cluster'. This term is used in both the logical and physical sense. That is, the term cluster when used in the context of computer

science generally brings to mind clustering in memory. However, the word cluster in the strict sense means to gather things together. In IDB, a cluster is a gathering of objects or a group of objects. An object cannot be in more than one cluster. Since IDB is built as a multi-user system, it also includes facilities for concurrent access control. Clusters are important in maintaining the access control. "The cluster is the unit of data transfer and the unit of locking for control of concurrent access" [Pe91d, p. 3].

Persistent Data Systems describes an IDB database as "a set of objects connected by references" [Pe91c, p. 6]. Since a cluster is a group of objects, then an IDB database is simply a set of clusters connected by references. There are two kinds of references in IDB: local references and cross references. The names themselves are quite descriptive. Local references are those references between objects of the same cluster. Cross references, on the other hand, are between objects belonging to different clusters. [Pe91d]

Access to clusters is gained by opening a transaction on the cluster. It is at this time that the cluster is read in from secondary storage into main memory. Local references are made by using pointers to the object being referenced. Cross references may also use pointers if the cluster containing the object they reference is already present in memory (i.e., a transaction is also open on that cluster). The other possibility is that the object being cross referenced is not in main memory. This case is resolved by using unique identifiers that IDB issues to every object. [Pe91d]

IDB guarantees that every object created will have a unique identifier. This identifier is not only unique to the cluster, system, platform, etc.; it is unique universally. That is, an object created by IDB on any platform can be ported to any other platform and still be guaranteed that the identifier is unique. This allows different platforms networked together to share IDB files without object identification problems. Thus, "no two objects

will ever have the same identity and an object will maintain its identity even when moved from place to place" [Pe91d, p. 4].

Several identifiers are needed to accomplish this: master identifiers (MID), cluster identifiers (CID), object identifiers (OID), and local identifiers (LID). During installation, a unique MID is set up that is assigned to each copy of IDB by Persistent Data Systems. When a cluster is created, the MID is combined with further numerical identifiers to form the CID. Finally, the OID is composed of the CID and the LID which uniquely identifies the object within the cluster. [Pe91d]

Each cluster is described by an Interface Description Language (IDL) structure. The structure contains all the necessary types to describe the objects of a cluster. In other words, it contains the schema for the cluster specified in IDL. The IDL structure is really the "description of structural constraints on data" [NMSW83, p. 7]. The fundamental IDL model is predicated upon the directed attribute graph. An example of a directed attribute graph is given in Figure 14.

A directed attribute graph is composed of a set of typed nodes that possess a set of attributes. The type of a node determines the particular set of attributes it will contain. The attributes are either a primitive value (the value is embedded in the node and is either boolean, integer, or rational) or a node-value (a pointer to another node). In Figure 14, the attributes with node-values have pointers which are the directed edges in the graph pointing to other nodes. Each graph must have a root node that allows all other nodes to be reached by following some path along different attribute edges from the root to the nodes. "An IDL structure specifies a related class of attributed directed graphs by listing the set of node types, the attributes of each node type, and the type of the root" [NMSW83, p. 7].

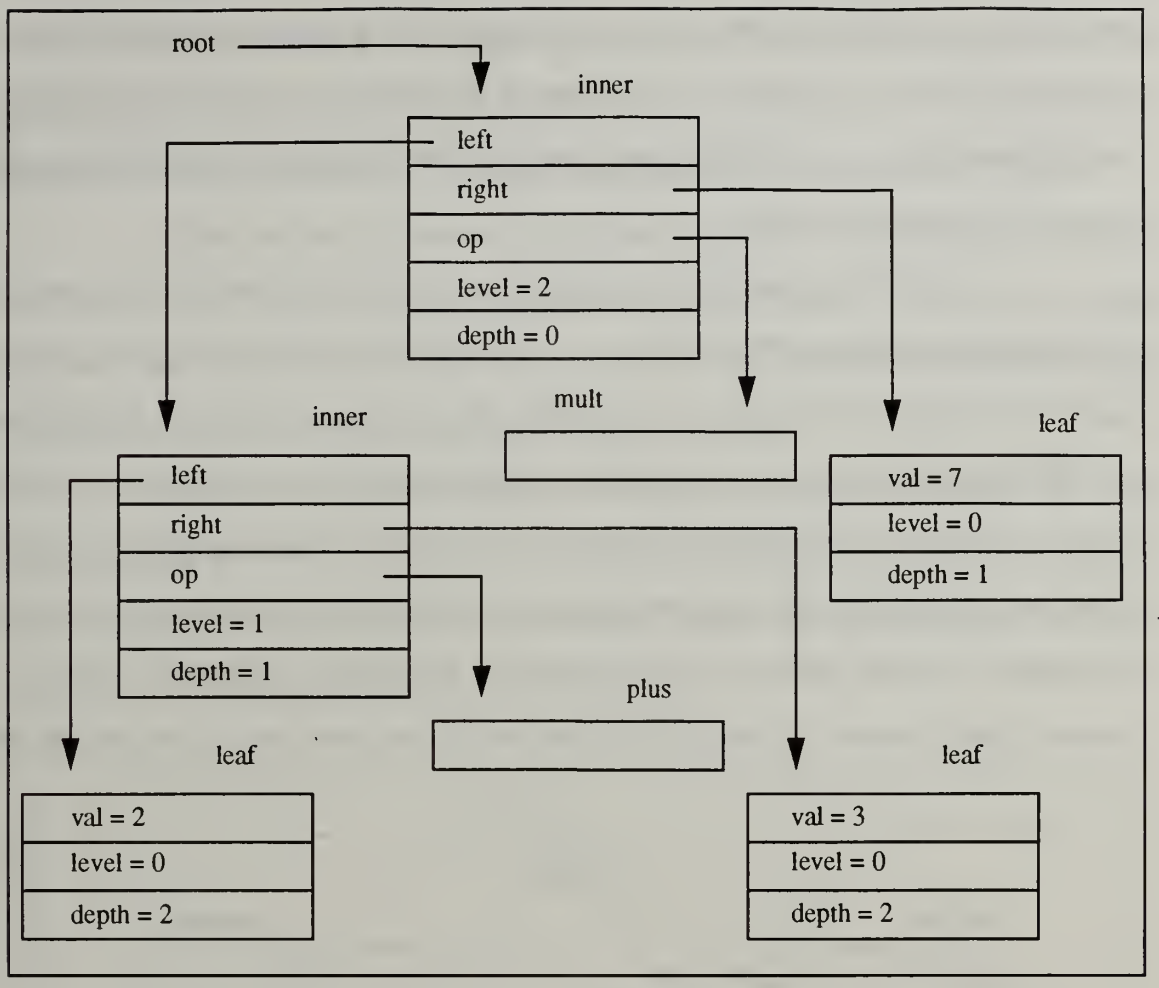


Figure 14 An Example of a Directed Attribute Graph [NMSW83, p. 8]

c. Nodes, References and Attributes

Every IDB object must have a type and can only be in one cluster. The types that an object may have are defined through the class structure that is delineated in the cluster schema. In IDB, only classes which have no subclasses can be instantiated as objects. These classes are called nodes and have the type node type. In the class hierarchy, the nodes are the leaves of the class structure. This is something of a departure from the standard object-oriented concept of a class since only leaf classes can be instantiated.

However, any class that is not a leaf can always be made a leaf (in a logical way) by instantiating a subclass of the class to only inherit all attributes and methods of its superclass. That is, it is a leaf which is a copy of its superclass except that the superclass has subclasses and the leaf (by definition) does not. Other class types that are not node types are said to be strict class types.

Consider the graph presented in Figure 15, which is derived from the example IDL schema (included as APPENDIX A) from the IDB User's Manual [Pe91d, p. 32]. It is clear from the graph that the leaf nodes are ptrain, ftrain, fplane, and pplane. These classes are IDB node types and may be instantiated to IDB objects. The other classes are used to build the attributes and methods of the node types but may not be instantiated into objects. These strict class types delineate attributes and methods that are common to all of their descendants [New86]. IDB also supports the idea of multiple inheritance. Thus, in our example, ptrain inherits all of the attributes and methods of class types passenger and train.

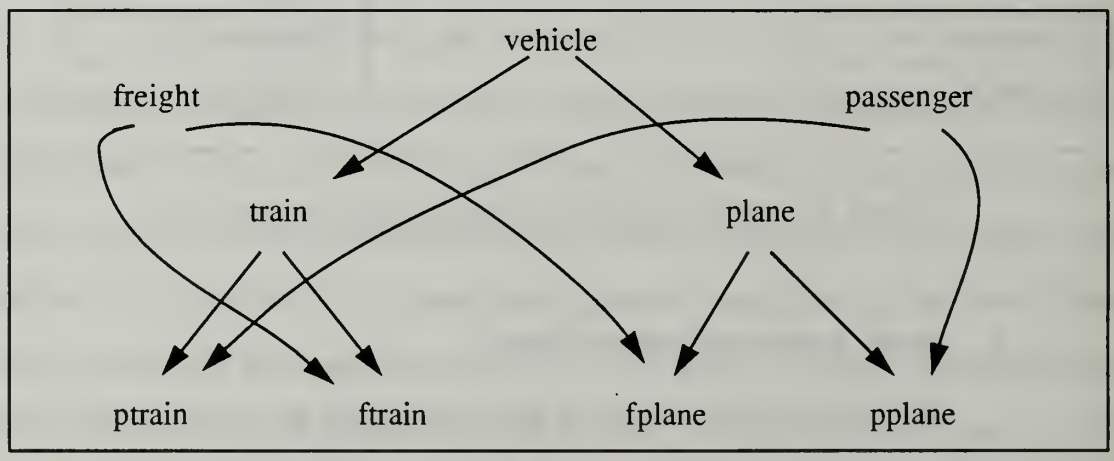


Figure 15 A Class Hierarchy

Newcomer likens the IDL node type to the idea of a record type in other languages[New86]. Record fields are similar to a node's attributes[New86]. There are two kinds of references which connect objects: local references and cross references. All

references must also have a type; however, unlike objects which can only have node types, references can have the type of any class in the cluster. A references of a certain type can point to any class of that type or any of its descendants (if the class is not a node). Thus, a reference of type train, in Figure 15 above, could point to a train, a ptrain, or a ftrain.

From the root reference of any cluster, all objects within that cluster must be reachable by following some path along references in the cluster. Two objects may have attributes that reference the same object. In this way, references facilitate sharing. References are also allowed to create cycles within a cluster.

There is another possibility for the type of a reference: universal types. The types discussed up to this point have all been associated with a particular cluster. Universal types allow references to refer to an object in any cluster. The universal types and their relation to each other is shown in Figure 16. [Pe91c]

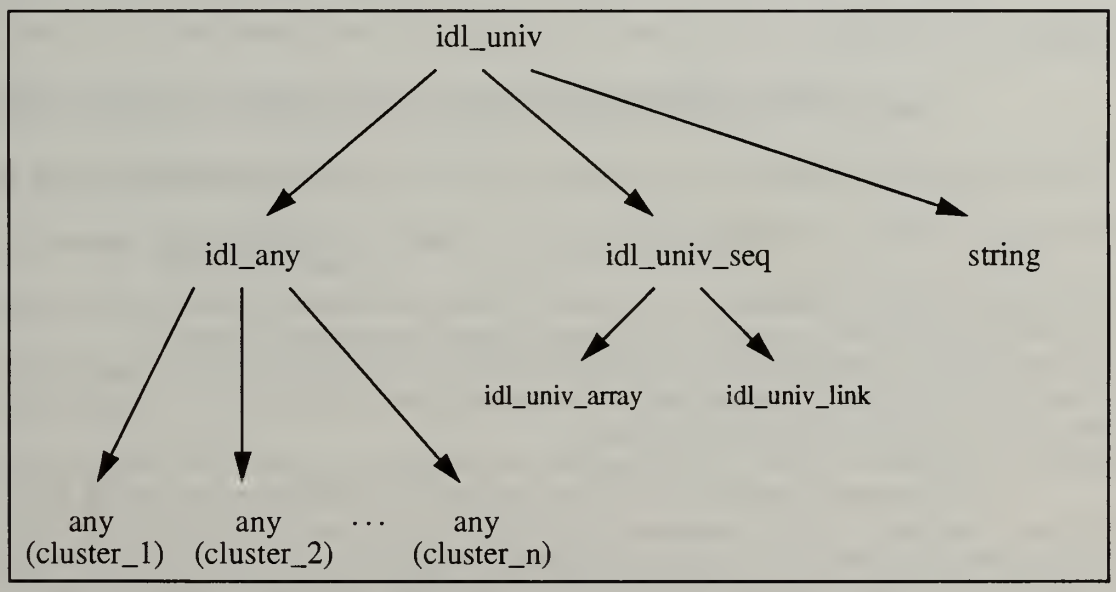


Figure 16 Universal Types [Pe91c, p. 36]

d. Transactions

Since IDB is a multi-user system, concurrency control must be maintained. Transactions are used to implement concurrent access control to clusters. Additionally, transactions guarantee that cluster updates are all or nothing. That is, the entire cluster is updated or nothing in the cluster gets updated. Thus, cluster modifications are 'atomic'. [Pe91c][Pe91d]

Transactions allow users to read and write clusters. Access to a cluster is determined by one of four types of transactions: write, create, read, and examine. To modify an object, a write transaction must be open. Examine and read transactions allow different forms of parallel access to a cluster by one or more user. New cluster values are created by the create transaction. A series of 'roll-back' points may be established by nested write transactions. An arbitrary number of write transactions may be nested within other create or write transactions. Large and complex modifications are facilitated in this manner. [Pe91d]

Reading and traversal of cluster objects and the ability to use those objects and their associated attributes to set values of browser variables is possible during read transactions. While a read transaction is open on a cluster, no write or create transaction on that cluster is allowed. Similarly, only while no write or create transaction is open, may a read transaction be opened. This guarantees that the cluster being viewed in memory is the same as that cluster on secondary storage.

Examine transactions are similar to read transactions except that it does not stop a write transaction from being opened and/or committed on the same cluster. Thus, the cluster being viewed in memory may be different (older) than that cluster on secondary storage. This transaction can be opened at any time. It allows a greater amount of

concurrent use of the system. Clearly, this kind of transaction should only be used when viewing the most current instance of the database is not necessary.

Modification of cluster objects and their associated attributes is accomplished using either a create or write transaction. Modifications made to the cluster are not written back to secondary storage until the transaction is committed. If after modifying the objects of a cluster in memory, it is determined that the modifications are unwanted, then the transaction may be aborted. In this case, that cluster in secondary storage remains unchanged. It is important to note that during a write or create transaction, one or more examine transactions may be open on the same cluster.

A special case of the write transaction is the create transaction. Create transactions may only be opened during entry to a cluster. They are used if no cluster file exists on secondary storage or if the cluster file that does exist is no longer wanted. The create transaction creates a legal value to which the cluster root is set. Once the create transaction is committed, then the created cluster and any other objects created during the open transaction are written to the cluster file on secondary storage.

4. Other Systems

a. ONTOS/Vbase

Vbase, a single-user system, a product of Ontologic Inc. (now named Ontos Inc.) released in 1988, had several problems that caused it to flop. First, it had several language problems. The Type Definition Language (TDL) was a nonstandard language which was plagued by many inherent problems associated with nonstandard languages. TDL was used for schema definition and abstract object interface specification [HW91]. Their object manipulation language was an object-oriented extension of the C language, called a compiled procedural language (COP) [HW91]. In addition to impedance mismatch[16] problems produced by these languages, Ontologic had to do everything to

support the languages and the languages had not passed the test of time. Additionally, there was a lack of tools needed to work with the languages. [In89]

ONTOS is the successor to Vbase [AHS91]. It runs under UNIX, AIX, or OS/2 [In89]. The design goal of ONTOS is that the system should allow trade-off decisions between performance, formalism, and safety to be made by the user [AHS91]. The system allows the user to directly access objects through direct references. In this manner, the user may bypass system mechanisms, controlling low-level detail to enhance performance if deemed critical to performance. ONTOS also supports C++, overcoming the nonstandard language problem of Vbase[In89]. Additionally, it provides a class library to enhance its modeling power. It also has no need for a schema definition language since it operates on class definitions in C++ directly. Versioning, an alternative mechanism, a graphical browser, an integrated object SQL, and a multi-user capability on a homogeneous network also exist in ONTOS. [AHS91]

b. GemStone

The Servio Logic Corporation produces GemStone[17] [BMO89][HW91], a disk-based storage manager designed for commercial and engineering markets [BOS91]. The designers of GemStone surveyed object-oriented extensions to C, Pascal, and the OOPL Smalltalk before deciding to develop their own OOPL, OPAL, a modified version of Smalltalk-80 [HW91]. OPAL is GemStone's data definition language/data manipulation language, which is also used for general computations/queries [BMO89]. Since it only uses

---

16. When information must pass between two structurally and semantically different languages impedance mismatch may occur [HW91]. Database systems that use two different languages in their implementation often have one that is a procedural language (conventional) and the other a more declarative, higher-level language which results in a mismatch of two language paradigms [Kh91].

17. The current version of GemStone is Release 4.0

one language, OPAL, for programming, it bypasses the problem of impedance mismatch [HW91]. GemStone provides an interface to several procedural languages: C, C++, and Smalltalk [BOS91].

Visual Schema Designer (GS Designer) and Tool Suite are two graphical tools which are included with GemStone. The GS Designer allows GemStone class definitions to be modified, deleted, and created using a keyboard and mouse interface along with bitmapped graphics in a windows environment. The class graph[18] is the primary organizing principle of the GS Designer. Both a high-level application development environment and a visual programming environment are integrated by the graphical tool 'tool suite'. Motif and OpenLook are both supported by tool suite. [BOS91]

Recently, Servio Corp. introduced a collection of development tools that facilitate object-oriented financial, scientific, and manufacturing systems construction by application developers. This tool collection is called the GemStone Object Database Development Environment, Geode. Geode is comprised of four components: the forms designer, the visual program designer, the application designer, and the system programmer tools. The forms designer facilitates the construction of screens for database information display, update, and insertion. The visual program designer comes with a set of basic libraries, which may be extended by the user, and applications to be developed graphically without writing code. The products of both the forms designer and the visual program designer are combined into complete applications by the application designer. Finally, debuggers, cross-reference tools, graphical browsers, and performance profilers comprise the system programmer tools. [Sc91]

---

18. A named group of classes interrelated by various types of relationships (including generalization, association, and aggregation) is called a class graph. All GemStone class graphs have the predefined Object class as its root. [BOS91]

GemStone supports several relational gateways. That is, it supports SQL access to the external databases Sybase, Ingres, Oracle, and Informix. SQL query results may be viewed as objects using these gateways which provides interoperability with relational databases. The translation between the systems is facilitated using generic row (tuple/record) and relation classes that may be specialized by the user by defining subclasses these generic classes. [BOS91]

c. POSTGRES

POSTGRES[19] is the successor to the INGRES relational DBMS, and has been under development since 1986 under the leadership of Professor Stonebraker at the University of California, Berkeley [St91b][SK91]. POSTGRES is often listed under the heading of OODBMSs. However, it is more than that since it also supports knowledge management. Actually, it is an extension of the traditional RDBMS that supports complex objects; inheritance; methods and functions in the database; and contains both a complete rule system and an abstract data type system [St91b]. POSTGRES II, the successor to POSTGRES is in the design process and will try to manage main-memory data, disk-based data, and archive-based data in a unified, elegant manner [BOS91].

The objectives of the designers of POSTGRES are to [Da90]:

- provide better support for complex objects,
- provide user extendability for data types, operators, and access methods,
- provide active database facilities (alerters and triggers) and inferencing support,
- simplify the DBMS code for crash recovery,

19. POSTGRES stands for Post INGRES and its current version is 3.0 [RK][SK91].

- produce a design that can take advantage of optical disks, multiple-processor workstations, and custom-designed VLSI chips, and
- make as few changes as possible (preferably none) to the relational model.

E. PREVIOUS WORK

In 1988, Michael L. Nelson completed the fundamental work in this area. Nelson's primary goal was to design a Relational Object-Oriented Management System (ROOMS) that could be implemented in almost any commercial object-oriented database or any object-oriented programming language [Ne88]. Stephen C. Filippi expanded the work done by Nelson with the completion of his Master's Thesis, Implementing Relational Operations In An Object-Oriented Database, in 1992 [Fi92].

1. ROOMS

ROOMS is the foundation upon which this thesis is built. ROOMS is a feasibility study to show that the relational data model need not be discarded in moving to object-oriented systems, and to allow the additional capabilities of the object-oriented paradigm realization within conventional applications by removing limitations on data types [Ne88][NMO90].

The fundamental structure of ROOMS is almost as simple as the relational system it imitates. An object that is a collection of tuples (records) is a relation. An object that is a collection of fields is a record. Objects that are user-defined class instantiations are fields. In ROOMS, all records of a relation must have identical format. Distinction between complex and simple objects are not made: no data type limitations. [Ne88][NMO90]

To show that ROOMS is feasible, the five fundamental relational algebra operations (selection, union, set difference, Cartesian product, and projection), which constitute the basis for all other relational operations, were implemented in a LISP-based

OOPL, PC Scheme (PCS). However, PCS has no facilities for object persistence so the data in this implementation is lost from one session to another. [Ne88]

2. Implementing Relational Operations in Prograph

This work continues the concepts of ROOMS by implementing the five fundamental relational algebra operations in Prograph, an OOPL. The fundamental contribution of this work is that of object persistence. Even though Prograph is not an OODBMS, it contains primitive operations that allow for reading and writing database files to secondary storage and for complex data type manipulation [Fi92]. The natural step that follows is to actually implement ROOMS in a commercially available OODBMS, which is the thrust of this thesis.

III. DETAILED PROBLEM STATEMENT

A. GENERAL

The focus of previous chapters has been to provide the reader with a very brief introduction to the motivation and topic of this work, a discussion of fundamental concepts necessary for a more complete understanding of this thesis, and a point of departure for this and future chapters. The purpose of this chapter is to provide an overview of some limitations associated with RDBMSs and OODBMSs, along with a discussion of the rationale for a combined R/OODBMS approach. In general, the limitations of RDBMSs tend to be the strengths of OODBMSs and the limitations of OODBMSs the strengths of RDBMSs.

B. RELATIONAL DATABASE LIMITATIONS

RDBMSs use a very simple data structure, the idea of a table (rows and columns), and are based upon a strong mathematical foundation, predicate logic and set theory. Primarily because of these two factors, RDBMS became widely accepted for business applications over the older and more awkward hierarchical and network database technologies. However, RDBMSs have a severe limitation: the inability to deal with complex data. It is the RDBMS's simplicity, along with its mathematical foundation and its complex data limitation, that are basis for this thesis.

1. Simple Data Types

For years, RDBMS have provided excellent performance for traditional, well established business data processing applications. Their standard fixed collection of data types (integers, rationals, strings, etc.) were sufficient to allow RDBMSs to function well. Now, as more complex data needs have developed within the business sector, particularly

in the engineering arena, it has become clear that this simple collection of data types is no longer adequate. Although relational systems have not outgrown their usefulness as traditional business database needs will always exist, there is definitely a need for something to overcome their limited data types.

Some current RDBMSs provide the ability to include digitized pictures in their database applications. However, none provide the ability to include more complex data structures such as sounds, animated graphics, and extremely large and complicated inter-relationships among relations within a database, to name some of the primary ones. Although some relational systems have been extended to include some of these complex data types, they generally do not include inheritance, encapsulation, and other characteristics of the object-oriented paradigm.

As RDBMSs attempt to maintain more complicated data relationships, performance of the system is degraded. The normalization process used in designing the database schema to represent complex data cause many small relations to exist within the database. As queries are made of the database, many join operations are then needed to answer the query. Since join operations are very expensive in terms of performance, the system tends to provide poor performance. However, in OODBMS this problem is avoided since these complex relationships can be represented with complex objects that have an explicit link between component objects. Thus, the need for numerous joins is avoided and performance is increased.

When modeling real world entities, the relational model uses relations (flat objects) to represent them. During the normalization process, the relations in a database schema are further flattened out into a number of smaller relations. This decomposition of real world entities into smaller flat relations represents a loss of abstraction. For example, in the previous chapter there was a real world entity that was represented by the relation Officer in a military database. A database containing the Officer relation might also have a

relation called Dependents to represent the real world entities that are dependents of military officers. However, this is not a very good abstraction. A better abstraction for these two real world entities in a military database is to have the Officer relation contain one more attribute: Dependents. But, this attribute would be multi-valued which is not consistent with the requirement that all attributes in a relational database be atomic. Therefore, multi-valued attributes are not supported in RDBMSs.

Continuing with the previous example, it can be seen that when an Officer tuple is retrieved there is direct access to the dependent data also. However, if two relations were used, then a join operation would be needed to associate an Officer tuple with that officer's associated dependents. This is a very simple example; however, the reader can imagine that if the complexity of this type of situation is increased several orders of magnitude, then surely a RDBMS would provide poor performance in responding to queries of this nature.

2. Tuple Function

As stated previously, relational databases are based on the simple data structure of a table of values of simple types. Displaying the values of a tuple in a relational database system is generally straight forward since the values are generally simple text or numbers that the database can easily handle with ASCII characters. For traditional business data processing applications, this is certainly sufficient. However, databases that contain more complex attribute values (such as sounds, graphics, video, etc.) are not so simple.

Consider a database of machine parts that contains as an attribute a 3-D image of each part along with more traditional attributes, cost, part number, size, weight, quantity on hand, etc. If a user wants to display a the 3-D image of a particular part, then the part tuple retrieved requires some method of displaying this attribute since a general RDBMS that was not designed for this specific application will not be able to display this attribute. However, an OODBMS could handle this situation by having a Display method for Part

objects which would display the 3-D image. Since OODBMSs[1] allow objects to have methods defined for them, any OODBMS would suffice to construct and maintain this machine parts database. This is in contrast to a RDBMS that would have to be developed specifically for this application since the data does not contain any functionality that could be used to determine how to handle it.

3. Inheritance

Relations in a relational database lack the ability to define a new relation based on an existing relation. That is, a new relation cannot be created that has a schema that only lists new attributes to be added to the schema for an already existing relation. For example, assume that a relational database has a relation called Person that has the following attributes: name, weight, birthdate, color eyes, and color hair. Now we wanted to add a new relation, Officer, that has all the characteristics (attributes) of Person but also has attributes rank, branch of service, and unit. It is desirable to allow Officer to inherit the attributes of Person. The usefulness of inheritance is more evident if we consider that we also want to have another new relation, Enlisted, that has the characteristics of Person along with enlistment date, rank, ETS, etc. Now we have two relations that have the characteristics of Person but do not have to redefine the common Person attributes in each of the new relations. In a relational system, these common attributes cannot be inherited but each new relation has to have them explicitly included in their schema.

Using the object-oriented paradigm, inheritance is included by definition and supports code/schema reuse by allowing an object to be further specialized by the definition

1. It is realized that there is no standard definition for an OODBMS however, in all cases it is generally agreed that for a DBMS to be considered object-oriented that it must manage objects that have attributes and methods associated with them.

of a subclass that inherits all of the attributes of its superclass. In our example, both Officer and Enlisted would be subclasses of the class Person. Now, any instance of an Officer or an Enlisted will include the common Person attributes without having them explicitly defined in the class definition for Officer or Enlisted.

4. Impedance Mismatch

With many conventional database systems (relational, hierarchical, and network), there is generally the problem of impedance mismatch. That is, they generally have one language for data queries and another for data manipulation. In the relational case, a query language like SQL may be used but then the actual data in the database is manipulated using a conventional programming language. Thus, there is a mismatch by mixing the generally procedural conventional language with the more declarative query language along with their differing data structures [Kh91]. In OODBMSs, there is a closeness between data and programs where a single language has the expressive power and flexibility to allow both data queries and manipulation.

C. OBJECT-ORIENTED DATABASE LIMITATIONS

Many feel that OODBMSs are here to stay and that they are the next logical step in the evolution of database technology. However, there still exist two primary drawbacks to them which limit their acceptance as the answer to tomorrow's database requirements: (1) the lack of a theoretical basis and (2) no universally accepted standard definition. Neither of these limitations exist in relational database model which has played a large role in the overall acceptance of relational model. Thus, there is no reason why a more expressive and powerful model that can easily have relational databases mapped onto it and overcomes these limitations cannot become widely accepted for tomorrow's applications.

1. Mathematical Foundation

Any query language in a relational system must be relationally complete[2]. Thus, RDBMS users are guaranteed that their queries will be answered correctly by any RDBMS system since relational algebra is based on mathematical set theory and relational calculus is based on mathematical predicate logic [EN89]. Queries in any language can be reduced to a mathematical premise and be mathematically shown to be true. This is not the case with OODBMSs which have no such theoretical basis.

2. Standardization

Possibly more serious than the non-existence of a formal theoretical basis is the lack of a universally accepted standard definition, which may be considered one of the relational model's stronger features. Since relational databases have a universally agreed upon definition and underlying theoretical foundation, different commercial RDBMSs are functionally equivalent. Standardization leads to many advantages such as better support, portability among different systems, greater acceptance, common evaluation criteria, etc. The OODBMS community lacks anything close to a standard or even a generally accepted definition.

Not only is there no standard definition or specification, but the object-oriented community cannot even agree upon terminology [Ed91][Ne91]. It is even more difficult to define something when people cannot even agree on the terminology that is used to describe it. Thus, when a database or any other software product is said to be object-oriented, the user cannot be sure what that means. Since 'object-oriented' is a hot

2. Recall that relational calculus and relational algebra are equivalent. Thus, it is also correct to say any query language must have the expressibility of relational calculus or relational algebra. In either case, a language that satisfies this requirement is said to be relationally complete.

buzzword, it is found in advertising for products that may have little to do with the object-oriented paradigm. This brings to mind a quote used earlier: "I have a cat named Trash. In the current political climate, it would seem that if I were trying to sell him (at least to a Computer Scientist), I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented" [Kin89, p. 23]. The bottom line is that the object-oriented community suffers from definition overloading.

This lack of standardization has also been problematic in that there are no standard benchmarks upon which to evaluate OODBMS performance [St91c]. Portability between systems is poor since no single OODBMS data model exists along with a standard core set of operations [Ed91]. OODBMSs differ among themselves on whether they have a class library available and what classes are provided in the library if it exists. This puts the consumer in a quandary since a decision must be made based on terminology that has different meanings depending on who wrote it to select from very different available OODBMSs that may be very costly. Since each OODBMS is so different and the computer software industry is so volatile, the consumer may lose all support and have to turn to entirely new system if the company that produced his system goes under. Clearly, if OODBMSs had a consistent terminology and definition, then it would be much more widely accepted.

3. Relational Operations

Companies that produce OODBMSs generally think in terms of managing objects and not relations with their database. Thus, there are not any commercial OODBMSs that support relations and relational operations[3]. However, an OODBMS that does support

3. Some do support an object-SQL interface.

relational operations would be valuable in terms of increased compatibility and completeness. It has been shown that relational operations can and should be made a part of an object-oriented database [Fi92][Ne88][NMO90]. An OODBMS that incorporates these operations would benefit from increased credibility along with acceptability among relational database developers.

4. Other Problems

Other problem areas that cannot be overlooked include the inability for most commercial OODBMSs to interface with relational systems [St91a]. A better solution is the integration of both object-oriented and relational systems. However, in the absence of an integrated approach, some interface should be available. Finally, along with the powerful modeling capabilities of the object-oriented data model comes an increased difficulty in making changes to the database as requirements change in addition to design difficulties [Ed91].

D. A COMBINED SYSTEM

1. Desirable Properties

A combined relational/object-oriented system will eliminate limitations that each of these systems possess as a system by itself. It would allow a relationally modeled system to naturally model and define complex objects along with their behavior which would facilitate better performance in complex applications. This ability is not found in simple extended relational systems. Generally, they allow for the definition of complex objects but not for their behavior.

Relational classes as part of a combined systems class library are needed: database, relation, and tuple (or record) classes. Additionally, the five fundamental relational algebra operations (select, project, Cartesian product, difference, and union)

should be a part of all OODBMSs. Credibility and standardization of combined systems will be enhanced as a result.

2. Possible Approaches

Two primary approaches to a combined relational/object-oriented system exist. The first is to take an existing RDBMS and extend the system to include all of the concepts in the object-oriented paradigm. Once this has been done, the underlying system is still relational so the system should still have all of the functionality of a purely relational system along with the advantages of the object-oriented paradigm. This would be a difficult and arduous task. Additionally, since the underlying system is relational, it may not be as efficient as an object-oriented system at managing objects. The other approach is to take an existing OODBMS system and construct classes and associated methods that would allow the OODBMS to provide relational functionality within the system. Thus, the relational model could be used with this system. This approach is much simpler since it requires the construction of three classes and their associated methods as compared with trying to implement all of the complex and powerful concepts of the object-oriented paradigm.

E. WHY THIS APPROACH

"The relational model is a viable approach to organizing persistent objects in an object-oriented database" [NMO90, p. 319]. The real world consists of complex entities/ objects that are easily modeled with an object-oriented paradigm. A R/OODBMS exists with the best of both worlds; it reduces the limitations that the individual systems realize by themselves. Conventional applications in a R/OODBMS can be extended to include complex data, and new non-conventional applications may also be developed in the same system.

Original work done in the development of the ROOMS paradigm [Ne88] and later work done with a R/OODBMS as implemented in Prograph [Fi92] is extended in this

thesis. ROOMS was not a full R/OODBMS since it lacked the ability to store persistent objects. A R/OODBMS in Prograph was a logical next step in the extension of ROOMS to a full database system.[4] This work completes the proof of concept that began with ROOMS and culminates in its extension into a commercially available OODBMS, IDB. The system implemented in this thesis is a single R/OODBMS that can fulfill the requirements of both relational and object-oriented users.

4. This is because Prograph is an object-oriented programming language that supports the storage and retrieval of persistent objects in secondary storage through its built-in database primitives [Fi92].

IV. IMPLEMENTATION OF AN R/OODBMS IN IDB

An R/OODBMS in IDB, as implemented in this thesis, is the culmination of a proof of concept that began with ROOMS [Ne88]. In that respect, it is not intended to be a production/commercial system. This chapter describes the fundamental design and detail of the implementation of an R/OODBMS in IDB. Although, there may be 'better' or more efficient implementations, the system as described is functional and does provide a proof of concept. It should also be noted that the author has worked exclusively with the commercially available IDB system. That is, there were no special/trade tools provided by Persistent Data Systems for this effort.

A. THE SYSTEM DESIGN

As described in [Ne88], the fundamental design of ROOMS is quite simple. This thesis is modeled after that design. A database is comprised of a group of relations. A relation is comprised of a group of tuples. A tuple is comprised of a group of attributes. Database, relation, and tuples are all implemented as classes. Both database and relation classes are IDB node class types while the tuple class is a strict class type.[1] Thus, tuples have as descendents user-defined classes that, when instantiated, become the tuples comprising a relation. In this manner, relations differ by the type of tuples they contain.

While it is possible to create stand-alone applications in IDB, the R/OODBMS was created for use within the IDB browser. Thus, the IDB core, display, and browser interface were used in the implementation of the R/OODBMS. The R/OODBMS user interacts with

1. Recall that in IDB a strict class type has one or more descendents while a node class type has no descendents.

the IDB browser to view the relations of a particular database and to perform relational algebra operations on relations within a particular database.[2]

Figure 17 shows a picture of the IDB Browser Interface. Pull down menus are listed along the top of the window. Within the browser, the programmer/application developer may introduce their own menus. We opted not to use this feature; instead the commands that the R/OODBMS user needs are displayed in the Menus pane.[3]

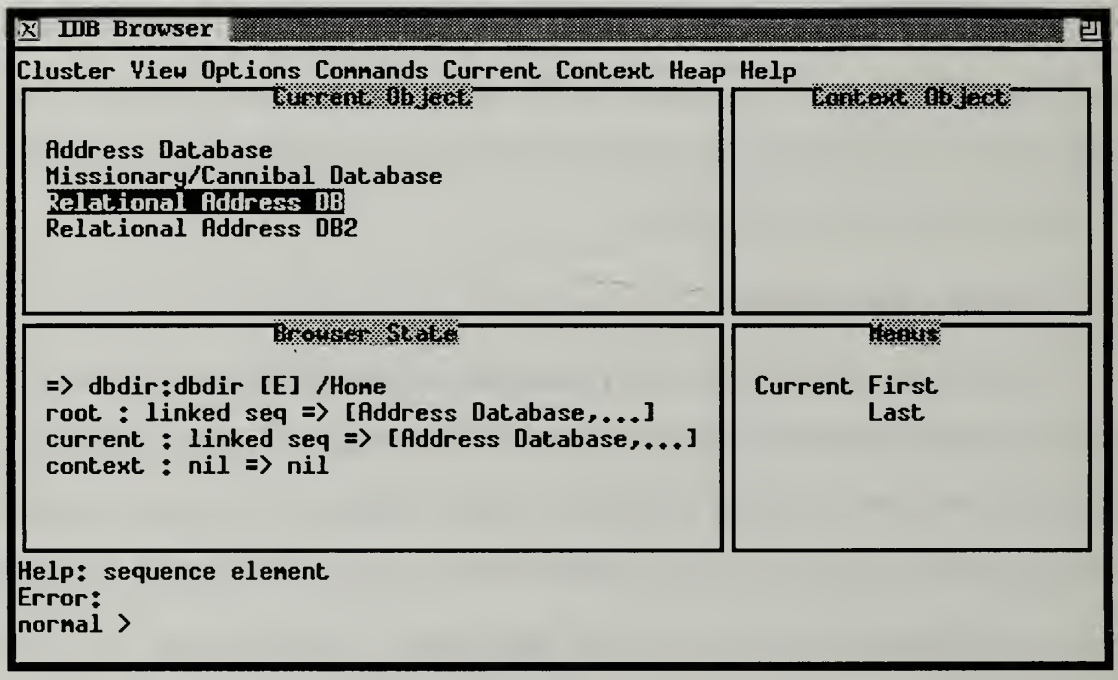


Figure 17 IDB Browser Interface

2. A database directory was developed to allow the R/OODBMS to manage more than one database. APPENDIX B contains the source code, the IDL schema and C implementation of the methods for the database directory cluster. APPENDIX C contains the IDL schema and C method implementations for R/OODBMS.

3. There are four panes within the browser window as shown in Figure 17: Current Object, Context Object, Browser State, and Menus. The user may have none, all, or any combination of these panes displayed at any time by making appropriate menu selections from the View menu commands.

Since the R/OODBMS is implemented as a proof of concept, it is not designed to have all of the facilities that are expected in a production system. Therefore, several assumptions are made about the R/OODBMS. It is assumed that the user will input correct information, in the correct syntax, for the R/OODBMS relational algebra operations and that the relations named as operands in the operation actually exist within the database. Additionally, all tuples within a relation must be instances of the same tuple descendent class. Thus, error checking is not provided consistently in all relational algebra operations. That is, different amounts of error checking have been provided among the operations in the hope of demonstrating the feasibility of detailed error checking without the overhead of implementing it for every operation. In two cases, detailed error checking has been implemented to demonstrate the feasibility of doing it for all operations, as will be described shortly.

By the same token, the relational operations, as implemented in this thesis, are implemented in the simplest manner possible while still demonstrating the feasibility/proof of concept of a R/OODBMS. For example, you may recall from Chapter II that the selection condition of a selection operation is a boolean expression consisting of clauses of the form:

<name of attribute><comparison operator><constant value>;

or

<name of attribute><comparison operator><name of attribute>

where <name of attribute> is the name of an attribute of the input parameter relation; <comparison operator> is either =, <, ≤, ≥, >, or ≠; and <constant value> is any arbitrary number of clauses may be connected with the AND, OR, and NOT operators to form the selection condition expression [EN89]. In this implementation of R/OODBMS, only one

clause is allowed for the selection condition since allowing an arbitrary number of clauses is just repeated applications of one clause in its simplest form.[4]

B. ORIENTATION TO R/OODBMS

This section is intended to familiarize the reader with using the R/OODBMS. It is not intended to be a complete user's guide. However, it should provide sufficient information so that a user could use the source code in APPENDIX B and APPENDIX C on a platform supported by IDB, translate the IDL schema, compile, and run the R/OODBMS.

1. The Database Directory

The database directory allows multiple R/OODBMS databases to be managed in IDB. If IDB is run with the following command line entry, then IDB will open the database directory cluster and display the browser window shown in Figure 17: 'idb -c dbdir -t dbdir'.[5] In Figure 17, there are four databases in the directory: Address Database, Missionary/Cannibal Database, Relational Address DB, and Relational Address DB2. If one of the databases is selected, say the Relational Address DB, then the browser window would appear as shown in Figure 18. Here the Relational Address DB object is the current object and the directory (sequence) of databases becomes the context object. To enter this database, select the enter command from the Menus pane.

---

4. Of course, there are more efficient methods of implementation than just repeated applications.

5. The name following the -c flag (dbdir in this case) in the command indicates the home cluster that will be used when IDB begins running. The name following the -t flag (dbdir) indicates the home cluster type. In this case, the home cluster is dbdir.c and its type is dbdir.

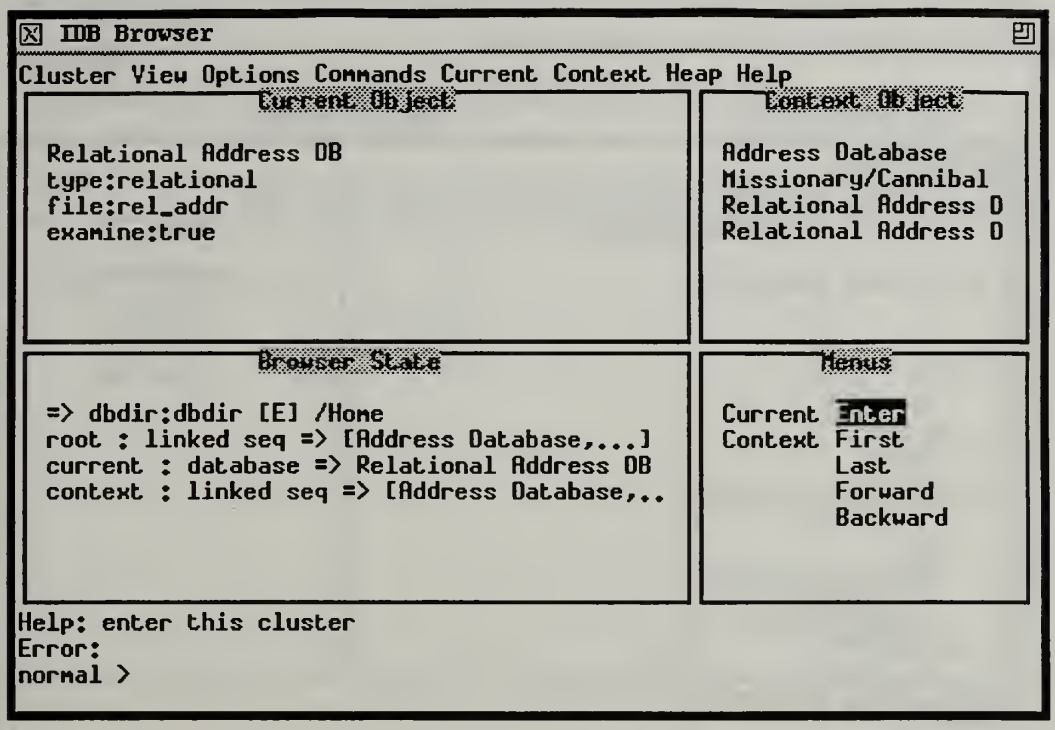


Figure 18 Entering a Database

2. Inside a R/OODBMS Database

Entering the Relational Address DB database, IDB opens the database cluster and displays the relations within the database in the Current Object pane as shown in Figure 19.[6] Now, any of the relations can be opened. For example, if the pt1 relation is selected then the Context Object pane will display the list of all relations within the database and the Current Object pane will display short form of the tuples within the relation pt1 (see Figure 20). Since each tuple within a relation is also an object, a tuple can be selected in which

6. In the interest of saving space and enhancing readability, interesting panes only will be shown in Figures.

case the list of tuples would move to the Context Object pane and the particular tuple selected is displayed in the Current Object pane.

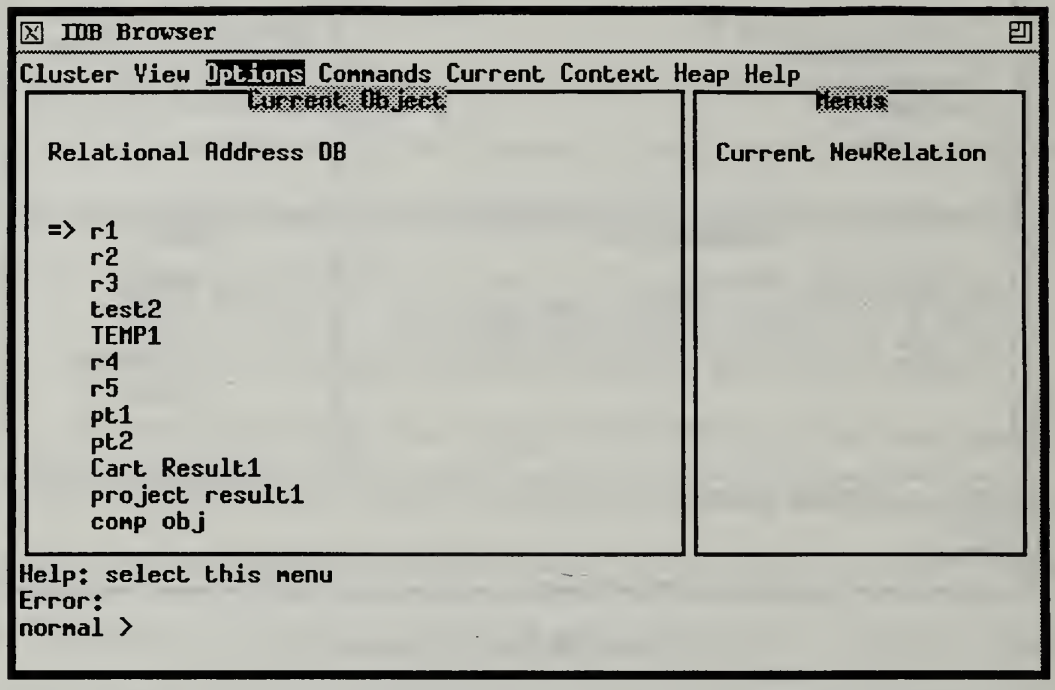


Figure 19 The Relational Address DB

Notice that as different objects are displayed in the Current and Context Object panes, the commands available in the Menus pane change. At any time, the Menus pane will display those methods associated with the objects being displayed in either of the object panes as long as the application programmer has specified them to be browser visible; there may be methods that are required by an object for interaction with other objects only that the application programmer decides the application user should not have access to (i.e., they are not browser visible).

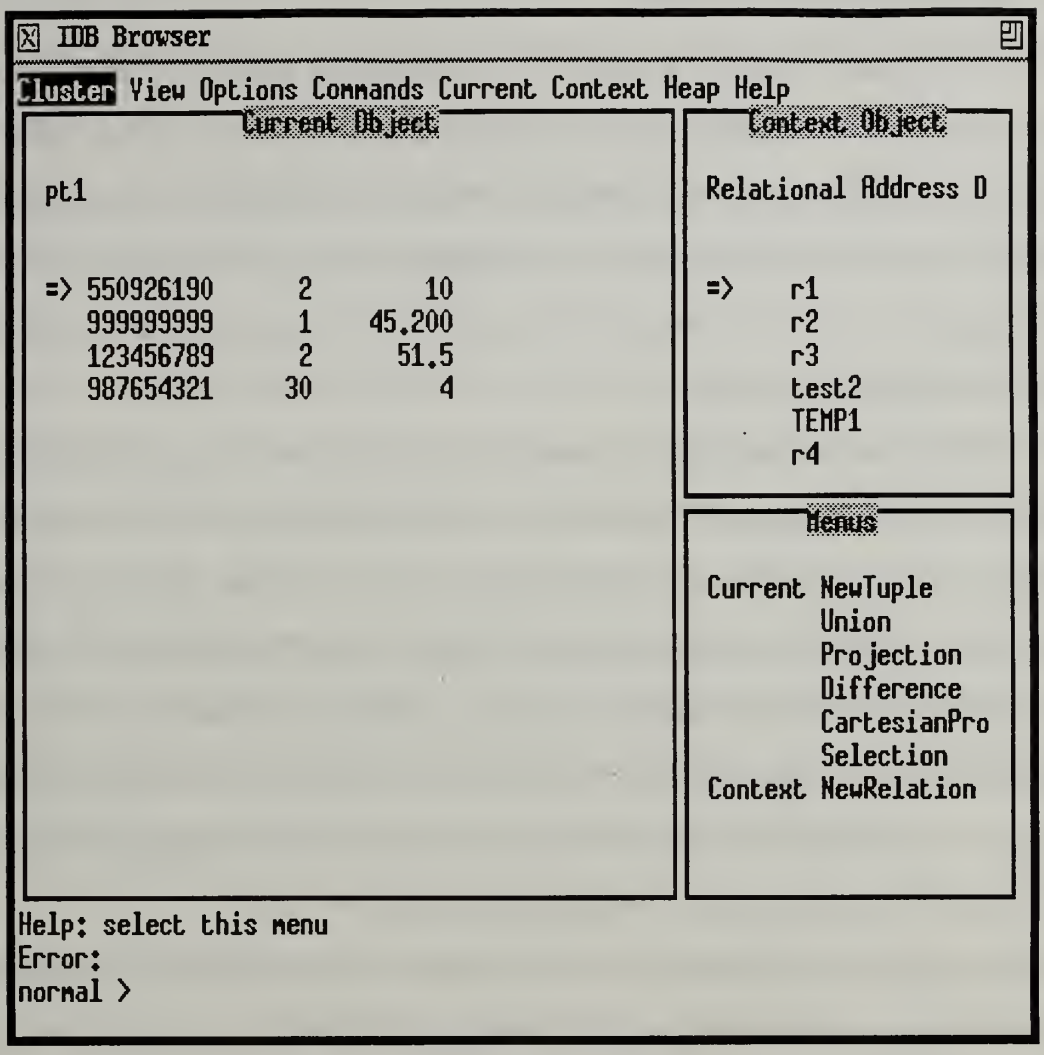


Figure 20 The Relation pt1

When any of the relations within a database is displayed in the Current Object pane, the Menu pane commands associated with the Current Object include the five fundamental relational algebra operations that are implemented within the R/OODBMS. Selecting any of the relational algebra operations brings up an associated pop-up window that allows the particular operation to be expressed in a query. The details of each of the relational algebra operations are discussed in the next section.

C. RELATIONAL METHODS

As discussed in Chapter II, there are five fundamental relational algebra operations from which all other relational algebra operations can be constructed: union, difference, selection, Cartesian product, and projection. These five operations are implemented as methods for the node class relation in the R/OODBMS. The difficulty in implementing these operations is directly related to the structure of the resultant relation as compared to that of the operand relation(s).

With the union, difference, and selection operations, the resultant relation has the same structure as the operand relation(s). Thus, the resultant relation's structure is already defined within the database and can be used to construct the resultant relation. In this sense, these operations may be considered simple. Cartesian product and projection, on the other hand, both yield resultant relations that have a different structure from the operand relation(s). Thus, the resultant relation's structure may not already exist within the database and either must be explicitly defined in the schema or dynamically constructed at run-time. Thus, these two operations may be considered difficult.

The details of each operation are discussed below along with difficulties encountered in their implementation and special implementation notes. Once one of the simple operations (i.e., result relation has the same structure as the operand) was completed, the other simple operations followed relatively quickly in their implementation. Selection, however, was more difficult than the other two since it required the implementation of default comparison operations: equal to, less than, and greater than.[7]

---

7. As discussed in Chapter II, there are six comparison operators: $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$. The last two are a combination of $=$ and $<$ and $=$ and $>$, respectively. $\neq$ is just the inverse of $=$. Thus, it is sufficient to implement only $=$, $<$, and $>$.

Before continuing, it is important to discuss the two possibilities for inserting tuples into the resultant relations: (1) a new tuple object can be created and then the values of the tuple can be copied into this new tuple, or (2) a reference to the tuple to be inserted can be used.[8] If the first approach is used, then the relations are independent of each other. That is, an update to any of the relations (operand or resultant) will have no effect on the other relations.

In the second approach, the relations become interdependent since they reference the same tuple relation. Thus, a change to the tuple in one relation will also be reflected in the corresponding referenced tuple in the other. This may or may not be desirable. A problem associated with this approach is the resolution between duplicate tuples in a binary operation. For example, there are two relations A and B, and we would like to perform the union operation. Relation A contains a tuple that is identical to one in B. Thus, it is a duplicate and must not be duplicated in the resultant relation, call it C. But, which tuple does C reference: the tuple in A or the tuple in B? They are both identical.

It would be best to allow the user to indicate in their query which of the two approaches is desired for a particular query. To show the feasibility of both approaches, some operations in the R/OODBMS were implemented using the reference approach while others used the copy approach. However, no operation was implemented with both approaches left for the user to choose from. It is clear, however, that if they can be implemented separately, then it is just a matter of additional programming and fine-tuning the user interface to add the ability to let the user choose.

The persistence of the resultant relation for these operations should also be considered. The resultant relation is not initially written to the database in secondary

8. That is, either make a copy of attributes in existing tuples or provide pointers to them.

storage, although it does appear in the list of relations for the database in the browser window (it only exists in main memory). If a write transaction is entered and committed, then the resultant relation will be written to secondary storage. If a write transaction is not entered and subsequently committed, then the resultant relation is lost when the transaction on the database is closed. Additionally, if a write transaction is entered and subsequently aborted, then the database cluster is read from disk again and any relation not previously stored on secondary storage is lost.[9]

Although a resultant relation only exists in main memory, it may be used in other queries just as any other relation within the database. It does not matter whether the temporary resultant relation has been committed to secondary storage, as far as the relational algebra operations are concerned - the resultant relations may be operated on as any other relation. For example, if relation R1 were unioned with R2 and the resultant relation TEMP_R3 were yielded, then a subsequent difference operation could be executed using TEMP_R3. It should be remembered, however, that even though these relations may be used in other queries, they are not written to secondary storage until a write transaction has been entered and subsequently committed.[10]

We will now discuss each of the 5 basic operations individually.

1. Union

The union operation is performed by the function union_op. It takes two relations, R1 and R2, and computes their union. The resultant relation is created by the function

---

9. Naturally, if a query is made during a write transaction, then the persistence of the resultant relation depends solely upon whether the write transaction is aborted or committed.

10. All resultant relations will be saved when the commit is executed. By the same token, if the write transaction is aborted then all of the resultant relations will be lost since a new transaction is entered after the abort which causes the database to be read from secondary storage.

init_temp_rel which takes R1 and R2 as its parameters and returns a relation with a unique name and some default values taken from R1. The syntax of the union operation (shown in the center of the pop-up window within the parentheses) is shown in Figure 21. Notice that there is no resultant relation listed in the query. This operation creates and names a resultant relation that will appear in the list of relations for the current database (this is in contrast to the project and Cartesian product operations where the resultant relation is named in the query, as will be discussed shortly). As previously noted, it would be desirable, in a production system, to have the ability to phrase a query with the option of providing a relation name or having a default name generated.

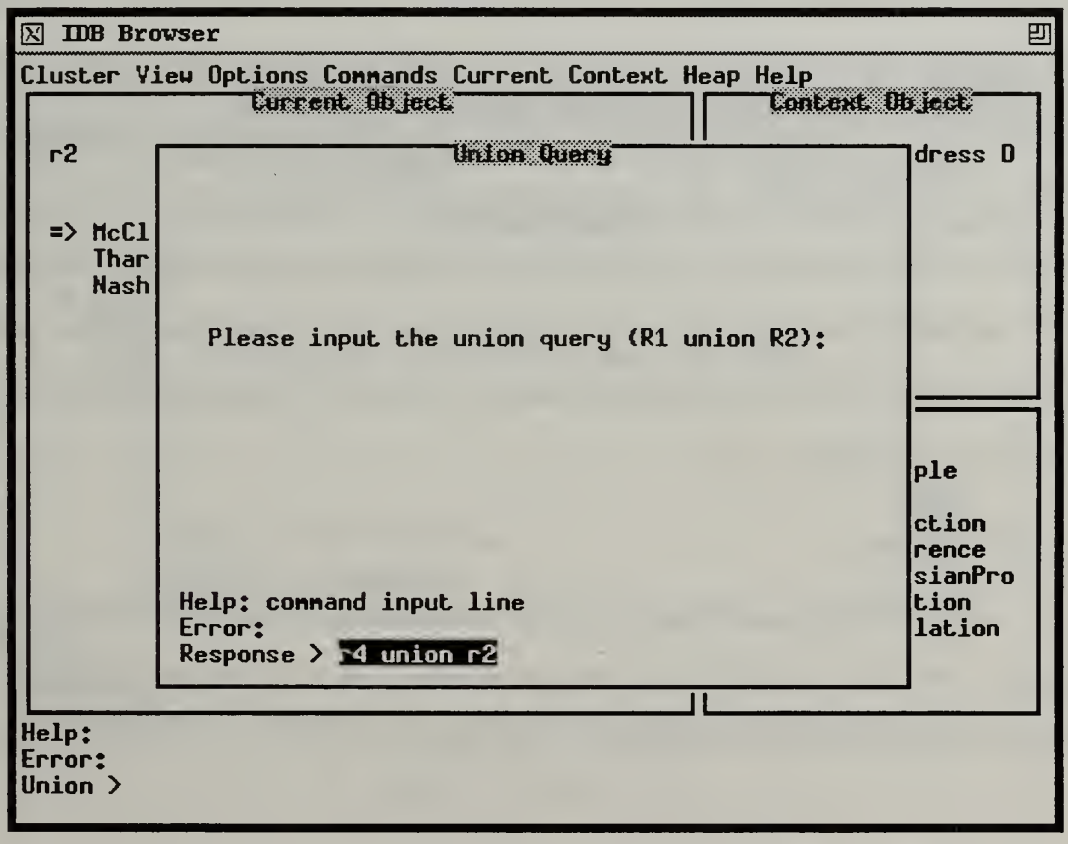


Figure 21 Union Query

The union operation, in contrast to the other operations[11], has extensive error checking in its implementation. The R/OODBMS first checks to make sure that the operand relations exist within the database. If both relations are found in the database, then they are checked for union compatibility. Provided that the relations exist and are compatible, a temporary relation is created by the init_temp_rel function and subsequently its tuples are inserted, ensuring no duplicate tuples. Tuples are inserted by reference and in the case of inserting a duplicate tuple, a reference is made to the tuple from relation R1.

The function used to check for duplicate tuples is the equal_to method which is defined for the class tuple. As such, it has a default implementation that only checks to see if the tuples reference the same object. This method will most likely have an over-riding user-supplied implementation for each tuple type. The prototype/signature for the method is boolean equal_to(tuple, tuple, index)[12]. That is, the function takes three parameters, two tuples and an index, and returns a boolean. Each of the tuples must be of the same type. When the value of the index is 0, then the equal_to function compares the each entire tuple, all the attribute values. Otherwise, index is greater than zero and is an index to the attribute within the tuples that is to be compared (selection operation). This function is also used in the selection operation.

2. Difference

The difference operation is implemented in a manner similar to that of the union operation. The function set_diff_op takes two relations and computes their difference. The resultant relation is created in exactly the same manner as in union_op. Set_diff_op also has

---

11. The difference operation has the same level of error checking as the union operation.

12. This is the actual C function signature/prototype that shows the function equal_to has three parameters (tuple, tuple, and index) and returns a boolean to the caller of the function.

the same level of error checking as union_op. The main difference between its implementation and that of union_op is in the insert decision for tuples that will comprise the resultant relation.

In the union_op function, all of the tuples from R1 are inserted into the resultant relation. Afterwards, the tuples from R1 are compared with those already in the resultant relation and if they are not duplicates, then the tuple from R2 is inserted into the resultant relation. In the set_diff_op, each tuple in R1 is checked to see if it exists in R2; if it does not, then the tuple is inserted into the resultant relation. Insertion is by reference here as in the union_op function. Figure 22 shows the difference query pop-up window with the correct syntax for the query.

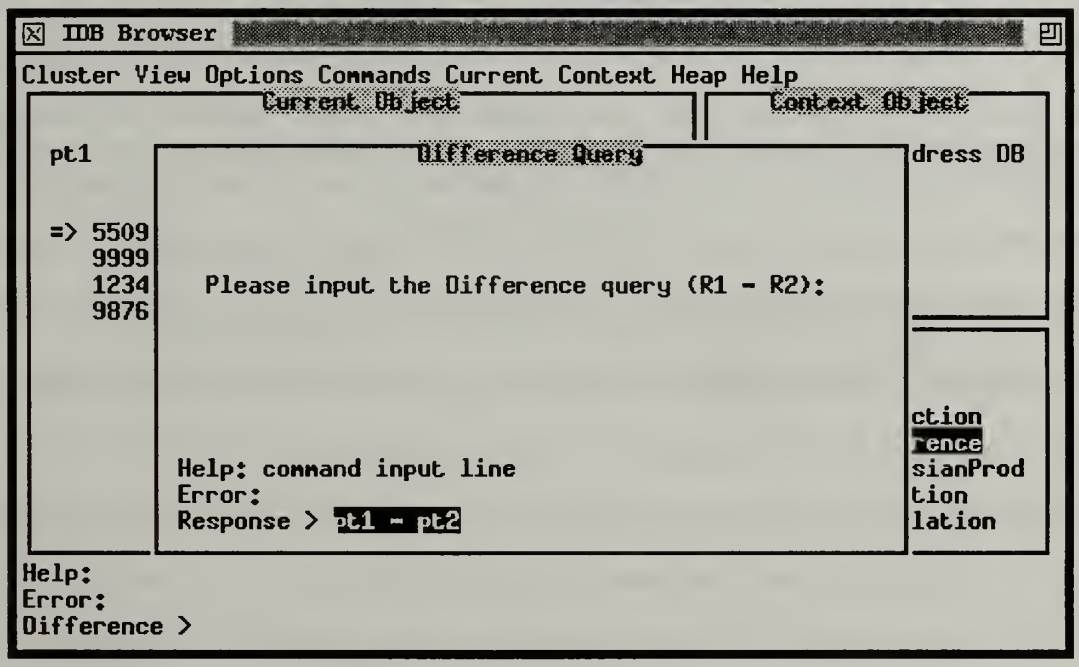


Figure 22 Difference Query

3. Selection

Of the three simple operations, selection was the most difficult to implement. It is a more complex operation since, even in its simplest form, it operates on a single attribute of a relation (which can be any one of the attributes in the relation's schema) and compares the value of that attribute for every tuple in the relation with a specified value to determine which tuples should be included in the resultant relation. Again, the resultant relation is created using the function init_temp_rel. This implementation, select_op, has the functionality necessary to demonstrate the feasibility of implementing a fully functional selection operation.

The general syntax for the R/OODBMS selection operation is:

<relation name> select <attribute name> <comparison operator> <attribute value>

where <relation name> is the name of the relation to be operated on, <attribute name> is the name of the attribute within the relation upon which selection will be based, <comparison operator> is a user-defined comparison operator (equal_to, greater_than, or less_than), and <attribute value> is the value to be compared with. Figure 23 shows the select query syntax. The particular query listed selects all tuples of pt1 where the attribute value of Hours Worked is greater than the Hours Worked attribute in Comp Object.

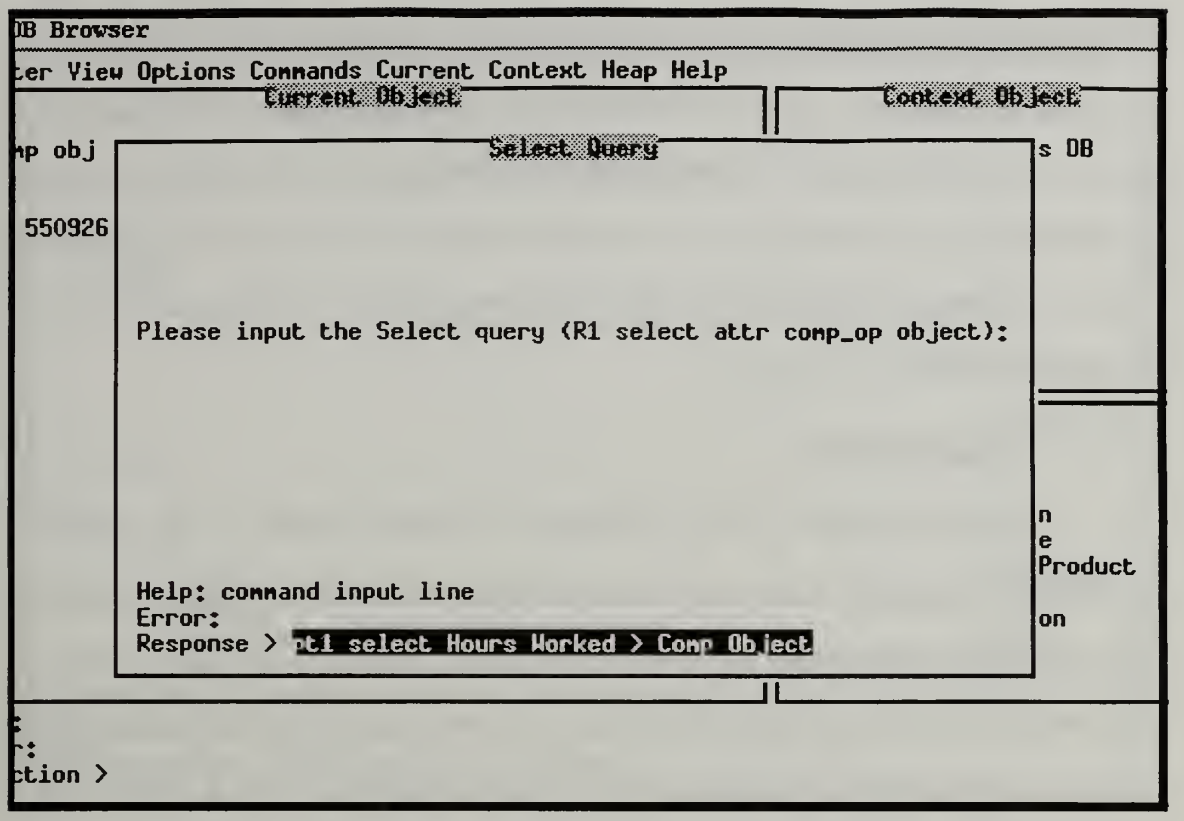


Figure 23 Select Query

It would be desirable to be able to list just a value for the <attribute value>; however, at a minimum it is necessary to be able to compare the same attributes of two objects. In R/OODBMS, the <attribute value> is a comparison object of the same type as the operand relation, R1. This comparison object can only have one tuple in it and may must have at least one of the attribute values entered, the one to be used for comparison. In Figure 23, Comp Object has the same type as pt1, thus it has the same attributes. The one tuple in Comp Object has 20 as the value for Hours Worked. Thus, the query is equivalent to select all tuples in pt1 where Hours Worked is greater than 20.

Select_op checks the database to ensure that both the relation to be operated on and the comparison object are both in the system. If they are, then the attribute name is used

to determine the index into the relation schema. If either of these conditions is not met, then the system brings up a pop-up window explaining that an error has occurred, what the error is, and allows the user to continue without crashing. Again, init_temp_rel is used to create and name the resultant relation for this operation. Finally, based on the comparison operator specified in the query, the appropriate comparison method is executed. The tuples of the operand relation are iterated through and those meeting selection criteria are inserted into the resultant relation by reference.

4. Cartesian Product

Of the two more difficult operations, Cartesian product is the simpler to implement as it operates on all attributes of the tuples in each of the operand relations.[13] Since the resultant relation has a different structure than either of the operand relations, the init_temp_rel function cannot be used. Instead, the user must provide the definition of the resultant relation within the database IDL schema beforehand. Now, a relation can be created within the database to hold the result of the operation. As a logical consequence, the relation that is to hold the result of the operation must be named in the query. Figure 24 shows the Cartesian product query pop-up window with the required syntax.

13. This is in contrast to the project operation where only a subset of the attributes will end up in the resultant relation. In a Cartesian product operation all of the attributes from each operand relation will be in the resultant relation.

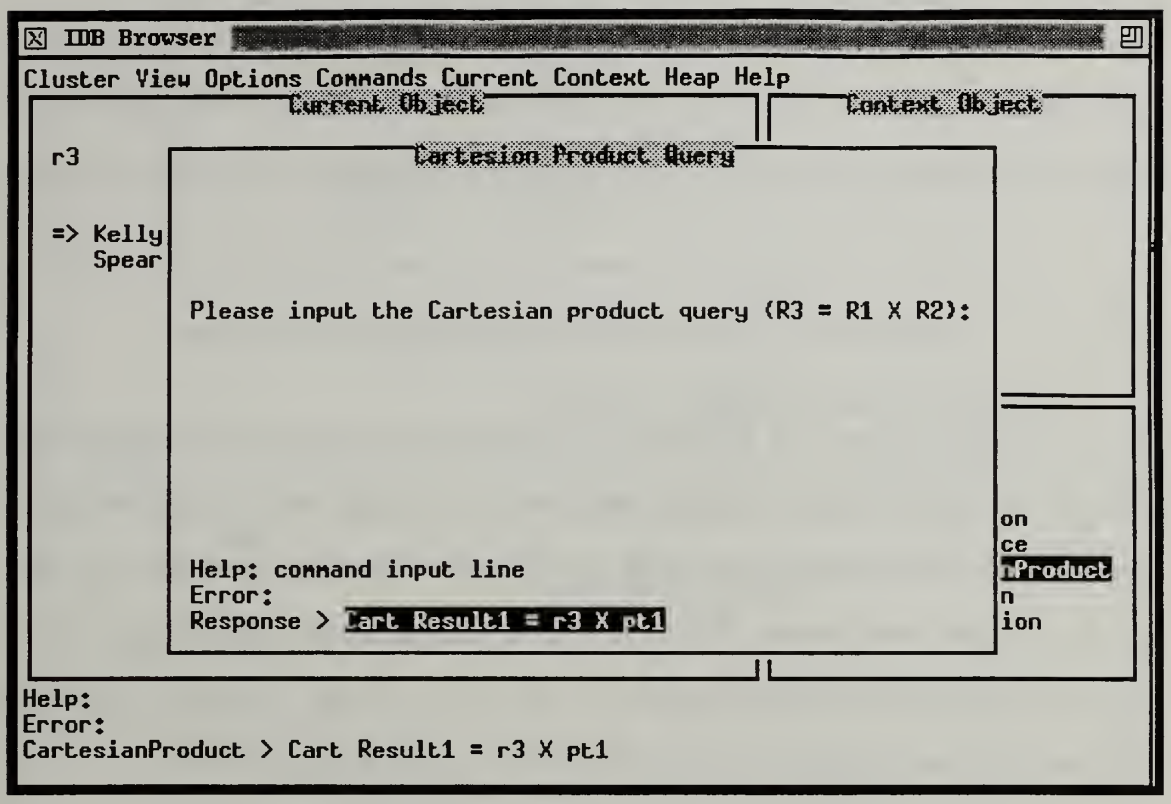


Figure 24 Cartesian Product Query

Defining the resultant relation's IDL schema is not difficult since we know that its schema is simply the concatenation of the two operand relation's schemas. No matter how complex the two operand relation's schemas are, the Cartesian product resultant relation will simple include the attributes as defined in both of the operand relations.[14] For example, suppose there are two relations, Employee and Assigned Project, with their respective tuple types defined with the IDL schema shown in Figure 25.

14. If multiple inheritance were available, the resultant relation could simply be defined as a subclass of each of the operand relations. As such, it would inherit the attributes of each of superclasses.

```
emp_tuple =>        person   :  person,
                    address  :  addr,
                    phone    :  phone_number,
                    widget   :  idl_univ;


proj_tuple =>       essn     : integer,
                    proj_num : integer,
                    hours    : rational;
```

Figure 25 IDL Schema for Employee and Assigned Project Relations

In our example, the attribute types of each of the tuples in the Assigned Project relation are simple: integers and rational. However, in tuples in the Employee relation, attributes have user defined (and provided) types and are, therefore, more complex. But, it is still a simple matter to create the schema for the resultant relation for a Cartesian product operation since all of the types (person, addr, phone_number, idl_univ, integer, and rational) are already defined within the database IDL schema (see Figure 26).

```
cart1_result_tuple =>   person   :  person,
                        address  :  addr,
                        phone    :  phone_number,
                        widget   :  idl_univ,
                        essn     :  integer,
                        proj_num :  integer,
                        hours    :  rational;
```

Figure 26 Example Resultant Relation Schema for a Cartesian Product Operation

Cart_prod_op implements the Cartesian product operation by first determining if the two operand relations and the resultant relation are in the database. If they do exist, then it is a fairly simple matter to take one tuple at a time from the first operand relation, concatenate it with each tuple of the second, and insert each resultant tuple into the resultant relation. The function insert_tuples is used by cart_prod_op to insert the tuples into the resultant relation.

In the query shown in Figure 24, r3 is a relation with emp_tuple type tuples, pt1 has proj_tuple type tuples, and Cart Result1 has cart1_result_tuple type. Figure 27 shows the relation r3 with its two tuples, Kelly and Spear. In the left side of Figure 27, all of r3's tuples are shown in their abbreviated form.[15] In the right side, a single tuple is shown in its entirety.

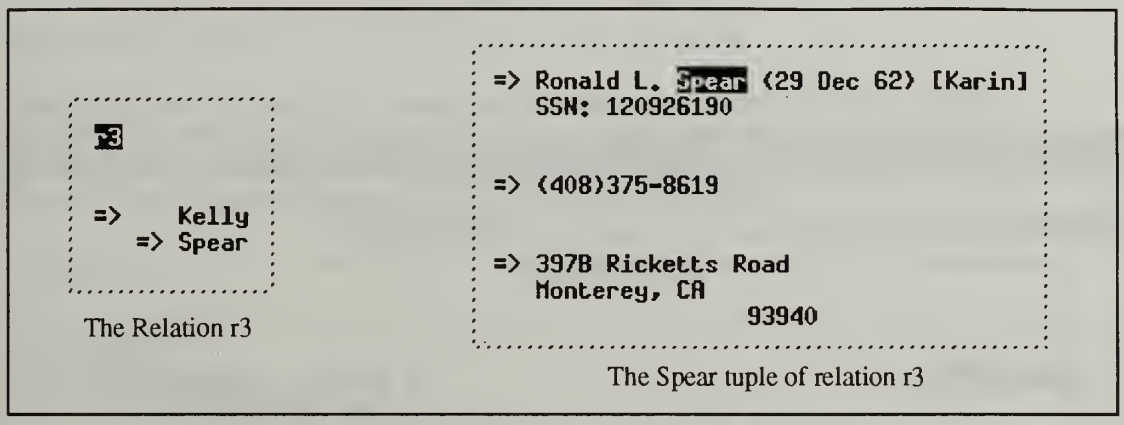
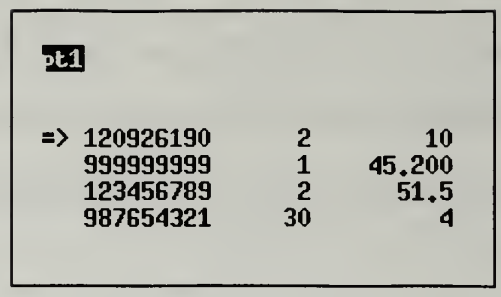


Figure 27 The Relation r3 and One of its Tuples

The relation pt1 is less complex and as such its abbreviated version has all of the information of each tuple. Therefore, there is no need to include a single tuple of pt1 in the Figure 28. If a single tuple were shown, there would be three values: an employee ssn, project number, and hours worked. However, these values would then be listed sequentially in a vertical fashion.

15. The abbreviated form only shows the last name of the person attribute. The predefined IDB functions idl_print and idl_key can have their implementations over-ridden by the user to present the relation and its separate tuples in any desired fashion. Figure 27 shows one such display.

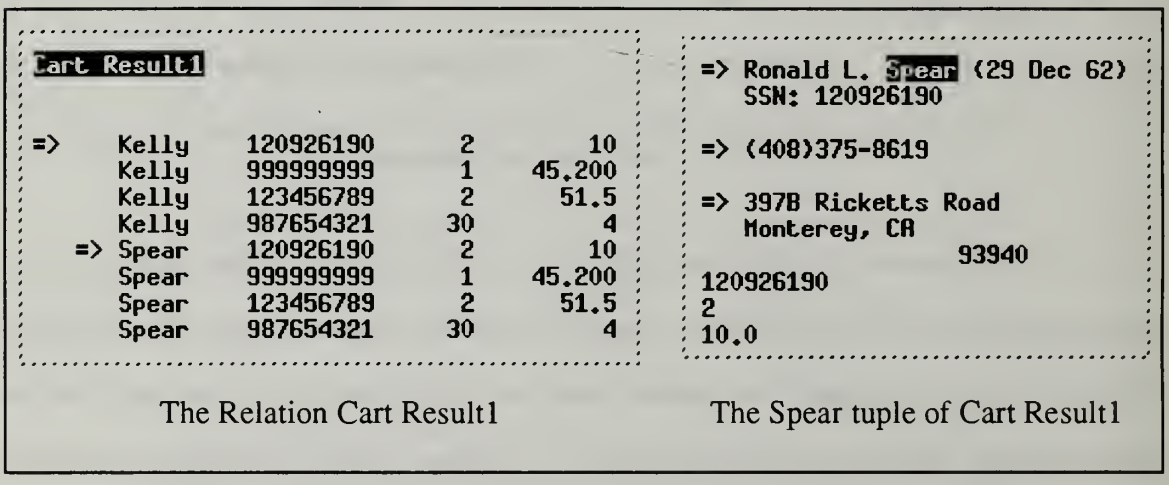


pt1

=>	120926190	2	10
	999999999	1	45.200
	123456789	2	51.5
	987654321	30	4

Figure 28 The Relation pt1

The result of the Cartesian product of relations r3 and pt1 is shown in Figure 29. The left side of the figure shows the abbreviated display of the relation Cart Result1 tuples, while the other side shows one entire tuple from the relation.



Cart Result1

=>	Kelly	120926190	2	10
	Kelly	999999999	1	45.200
	Kelly	123456789	2	51.5
	Kelly	987654321	30	4
=>	Spear	120926190	2	10
	Spear	999999999	1	45.200
	Spear	123456789	2	51.5
	Spear	987654321	30	4

The Relation Cart Result1

```
=> Ronald L. Spear (29 Dec 62)
   SSN: 120926190

=> (408)375-8619

=> 397B Ricketts Road
   Monterey, CA
                        93940
120926190
2
10.0
```

The Spear tuple of Cart Result1

Figure 29 Cartesian Product of r3 and pt1

5. Projection

Although the projection operation has a resultant relation with the same number of tuples as the operand relation, it is complicated by the fact that any combination of the attributes of the operand relation can constitute the schema of the resultant relation. As with Cartesian product, since the resultant relation has a different structure than that of the

operand relation, the user must provide the definition for the resultant relation in the IDL schema beforehand. Again, this is not a difficult task since the schema of the resultant relation is a subset of the attributes of the operand relation. Thus, all of the attribute types must already be defined in the database schema. Parsing the query is more difficult since any combination of the attributes can be specified in the query to be projected into the resultant relation. Figure 30 shows the browser pop-up window that accepts the project query and also specifies the query syntax for this operation.

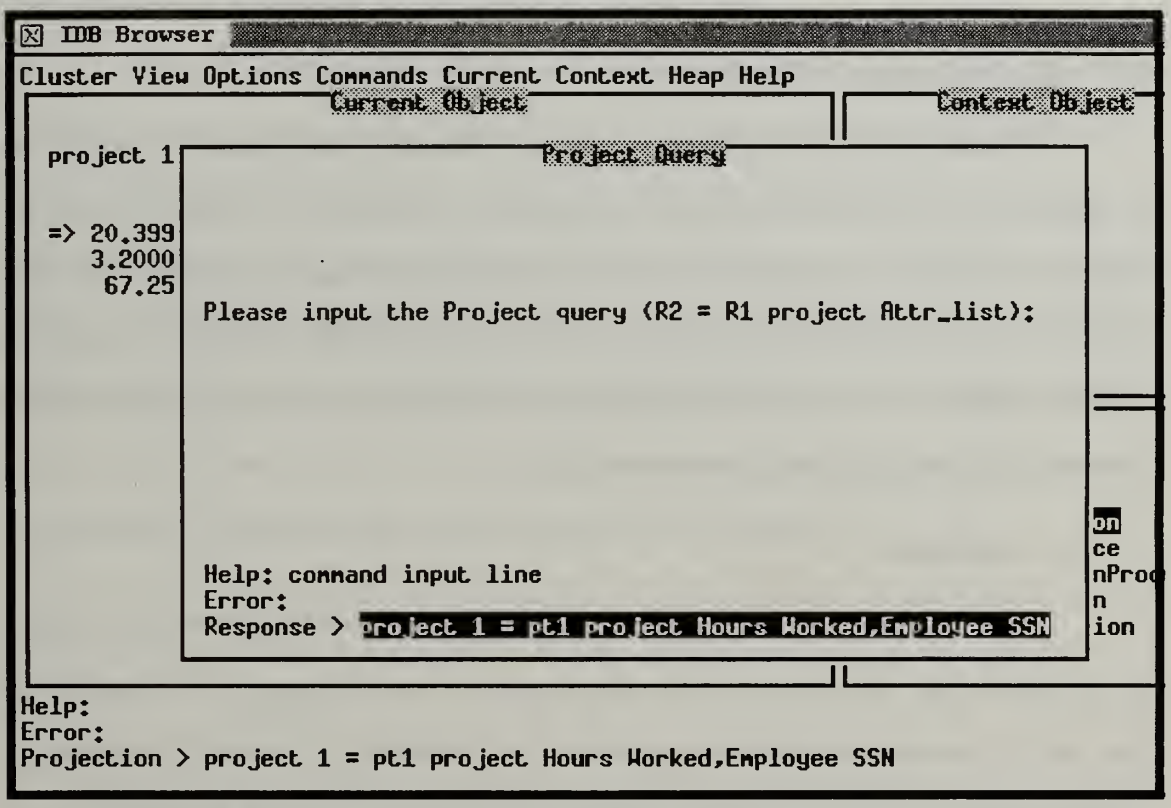


Figure 30 Project Query

Consider a relation with tuples of type proj_tuple as defined in Figure 25. If the user wanted to project the hours worked and employee ssn of each tuple, then a schema for the resultant relation would look like the one in Figure 31. The function project_op

implements the operation. Once it has checked to ensure that the operand relation and the resultant relation exist, it calls the function insert_fields to insert the appropriate attributes into the resultant relation.

```
project1_result_tuple   => hours :  rational,
                           essn  :  integer;
```

Figure 31 Example Resultant Relation Schema for a Project Operation

D. THE DATABASE CLASS

The database class is an IDL node class and is actually quite simple. Its sole purpose is to keep track of the relations within the database. As mentioned in Chapter II, each IDB cluster must have a root class from which all other classes can be reached. For the R/OODBMS, the database class is the root. Thus, all functions that implement the relational algebra operations start their search for the relations from this class. This coincides with the idea of a database being a collection of relations.

1. Attributes

The database class has only two attributes: name and relations. The name attribute is included so that when a database is entered users can see the name of the database that they are in. However, none of the operations in the R/OODBMS require this attribute. The relations attribute is very important since it is the means by which all relations in the database are reached. Its type is a sequence of relations where relation is another IDL node class. The sequence is a doubly-linked list.[16] All of the R/OODBMS relational algebra operations use the relations attribute to iterate through when trying to locate a particular relation within the database.

2. Methods

As with all objects in R/OODBMS, the predefined IDB core interface methods idl_key and idl_print are over-written.[17] Simply stated, idl_print displays an object while idl_key displays a short description of the object [Pe91a]. Their predefined implementations are functional but, of course, cannot anticipate the types of objects that will be displayed, much less the most pleasing format to display them in. In the following discussion of other classes these methods will not be discussed again although each has its own implementation for the idl_key and idl_print methods.

The only other method defined for the database class is new_relation which, as the name implies, creates a new relation and inserts it into the database. To create a new relation using the new_relation method, the user must first enter a write transaction. Otherwise, when this method is selected from the Menus pane, an initialize new relation pop-up window will appear but the user will not be able to view and edit all of the relation attributes necessary. If a write transaction is open when new_relation is selected, then the pop-up window will display all attributes of a relation ready for editing.[18] The user is responsible for initializing the relation attributes to valid values.

---

16. Sequences in IDB are either arrays (the default) or doubly-linked lists. Arrays are convenient for sequences with a static number of entries. However, every time the size of the sequences changes there is a great amount of overhead involved. Linked lists support sequences whose number of entries varies dynamically.

17. Both idl_key and idl_print can only be used with the display manager. An exception will be raised if attempts are made to invoke these methods without the display manager. [Pe91a]

18. The idl_print implementations for relations do not display all attributes during read and examine transactions. However, since the user needs the ability to edit all attributes, they are visible during write and create transactions.

E. THE RELATION CLASS

Codd's relational model is simple, representing databases as a collection of relations (or tables). Hence, relations are the fundamental building block of the model and as such the relation class of our R/OODBMS is the fundamental building block of the system. As with the database class, the relation class is an IDB node class type. Since a relation is a collection of tuples, it stands to reason that the attribute tuples is the most important attribute in this class.

1. Attributes

a. Relation_name

This attribute plays a greater roll in the R/OODBMS than does the database classes' attribute name. The relation_name allows different relations within a database to be easily differentiated. Of course, the database name within the database directory plays as great a role. The type of this attribute is an IDB string.[19]

b. Attribute_names

The relational schema is specified by the relation's attributes which are listed in attribute names. This attribute has a type that is a sequence of name where name is an object/class (an IDB node class) that has a single attribute, name. The name attribute of the name object is a string type. Thus, attribute_names is simply a sequence of strings. Since the schema of a relation is static, the default array sequence is used here.

---

19. An IDB string differs from a C string in that it has an additional null character at the end of the string. That is, there are two null characters at the end of an IDB string. IDB strings can generally be used in the same manner as C strings.

The degree of a relation is simply determined by the number of attributes. This is easily done by checking the size of the array that comprises attribute_name. Two relations are said to be union compatible if each is of the same degree and if corresponding attributes have common domains. Thus, attribute_names is used by both the union and difference operations to determine if the two operand relations are union compatible.

The project and select operations are the only other R/OODBMS relational algebra operations that require the attribute attribute_names since they operate on a subset of the attributes and single attribute of the operand relation, respectively. Specifically, attribute_names is used to determine if an attribute(s) in a query is in the operand relation or not. Additionally, it is used to determine the index (position within the relational schema) of an attribute.

c. Attribute_types

Attribute_types is of the same type as attribute_names, however, it is used only in support of the union and difference operations to determine if operand relations are union compatible. As stated above, union compatible relations have corresponding attributes with a common domain. Having a common domain means that the attributes have the same types. Thus, once two relations are determined to be of the same order, their attribute types are checked to ensure that they have a common domain.

d. Tuples

The tuples attribute within a relation is a sequence of tuples where tuple is an IDB strict class type. Thus, tuple has subclasses which are referenced by this attribute.[20] Thus, IDB allows a relation to have a sequence of tuples where some of the tuples reference

20. An attribute in IDB that has a strict class type can reference that class or any of its descendents.

one subclass type of the IDB strict class tuple while others reference a different subclass. However, if allowed in an R/OODBMS, then this sequence of tuples should not be called a relation, as this would allow various tuples to have a different number of attribute values and/or have different domains for corresponding attributes. Thus, all tuples within a relation must be of the same subclass of the IDB strict class tuple.

It is the tuples attribute that allows one relation to differ from another in structure. All of the descendents of the IDB strict class tuple are user defined. The user may define a subclass of tuple to have any desired structure. For example, the user may with to have an employee relation and a project relation where the employee relation contains personal information about the employee while the project relation contains information about different projects that the company has worked on. A relation should be constructed to model each. To create each relation, the user must first define the type of tuple that will comprise each relation: an employee tuple and a project tuple. Now, the new_relation method of the database class is used to create the new relations. Each relation should be given a descriptive name. Only employee tuples should be inserted into the employee relation and project tuples into the project relation. This restriction must be enforced by a production system; however, our proof of concept system assumes that the user will do this.

Since the number of tuples within a relation varies with time, the size of the tuple sequence is dynamic. Database updates, insertions, deletions, etc. generally necessitate a modification to the number of tuples within a relation. Therefore, the sequence of tuples that comprise the tuples attribute is a doubly-linked list which allows the sequence to dynamically grow and shrink at run-time.

e. Tuple_type

Tuple_type is of the IDB strict class type tuple. Thus, it can reference any of the user defined tuples that comprise relations within the database. When a relation is

created, the tuple_type attribute must reference an object that is of the same type as all tuples within the relation. Therefore, an object should be created solely for that purpose and then the tuple_type attribute needs to reference it.

As discussed in Chapter II, IDB invokes methods by the use of two different operations: idl_vop and idl_top. Idl_top is used to invoke a specific method implementation within the class hierarchy while idl_vop invokes the implementation of a method that is most closely defined for an object. For example, idl_print has a default implementation that is inherited by all classes. Consider the two tuple types employee tuple and project tuple, both of which have specific implementations for the idl_print method.[21] Both of these classes have tuple as their superclass and tuple also has a specific implementation defined for idl_print. Thus, there are now four implementations for the one method, idl_print: the default, the strict class tuple's, employee tuple's and project tuple's. Idl_top allows us to explicitly indicate which implementation of idl_print is invoked regardless of which object is being printed while idl_vop invokes the specific implementation closest in the hierarchy to the object that is to be printed. Thus, using idl_top, an employee tuple could be printed using the strict class tuple's implementation. However, if idl_vop were instead used to invoke the idl_print method, then the employee tuple's implementation would be invoked.

Tuple_type is used in an idl_vop operation to indicate which implementation of a method to invoke. This is used most often within the method new_tuple (create_tuple is its C implementation) to invoke the correct implementation of initialize_tuple. Each descendent of the strict class tuple must have its own user defined and specified implementation for initialize_tuple.

---

21. That is, along with their IDL schema definitions, both classes have redefined the implementation for the idl_print method.

f. Key

The default key used in the R/OODBMS is to check the value of every attribute within a tuple. Each relation has at least one superkey: the key that is all of the attributes of the relation. By definition, each tuple within a relation must be unique. Since a tuple is comprised of all its attributes, the set of all its attributes is therefore a key.[22] When this implementation is over-ridden the user may use this attribute to specify some other key to be used when comparing tuples.

2. Methods

Other than the relational algebra operations, idl_key method, and idl_print method that have already been discussed, the IDB node class relation has two other methods defined: new_tuple and check_union_compatability. Their names are quite descriptive of their function. As mentioned earlier, create_tuple is the C implementation of the method new_tuple. As with all of the node class relation's methods, new_tuple requires no user defined methods. However, it does invoke a user defined and provided method: initialize_tuple (a method of the strict class tuple). The function of check_union_compatability has already been discussed in the sections pertaining to the attributes attribute_names and attribute_types.

F. THE TUPLE CLASS

User defined tuples must be subclasses of the strict class type tuple. Of the three required classes that comprise the R/OODBMS, tuple is the only one that is not an IDB node type. As such, it cannot be instantiated into tuples (objects) that can be inserted into

22. This is not to say it is the only key. There usually exists other keys that are a subset of attributes comprising a relation.

relations. Instead, the user must provide the definition for tuples that are descendents of this class.

1. Attributes

Tuple, as a strict class type, has no attributes defined for it. There are no attributes that are common to tuples of every relation. Thus, it makes no sense to define attributes at this level since they would be inherited by all descendents.

2. Methods

Each method defined for this class has a default implementation that is very general. As a whole, most of the default implementations are so general that they tend not to be very useful. Users must provide their own implementation for each method defined for this class. Clearly, the implementation of the method initialize_tuple must be different for an employee tuple than for a project tuple since they likely have different orders and differing domains corresponding to their attributes.

a. Initialize_tuple

As the name implies, the function of this method is to return a newly created and initialized tuple. It has a tuple as its only parameter, creates a new tuple, initializes the tuple with valid values, and returns the new tuple.

b. Insert_fields and Insert_tuples

The first approach attempted in implementing these insert_fields and insert_tuples, which support the project operation and Cartesian product operation respectively, was unsuccessful. Consider the Cartesian product operation: to form the resultant relation each tuple of the first operand relation is concatenated with each tuple in the second. This concatenation process is continued for every tuple in the first relation. It therefore seemed logical to create a function called insert_tuple that would take as its

parameters two operands, the tuple to be inserted and the resultant tuple. Each user defined tuple used within a relation in the database would have their own particular implementation of the method.

For example, recall the tuple types defined by the schemas in Figure 25 and Figure 26. To insert the resultant tuples into the resultant relation for the operation Cart Result1 = r3 X pt1 (where the relation Cart Result1 has tuples of type cart1_result_tuple, relation r3 has type emp_tuple, and pt1 has type proj_tuple), the relation r3 would be iterated through a tuple at a time. The insert_tuple method for r3 would then be invoked to insert the r3 tuple values into the resultant relation tuple. Subsequently, insert_tuple would be invoked again however, this time for pt1 which would insert its tuple's values into the resultant relation tuple. Thus, one complete tuple of the resultant relation is complete. This would continue with the first tuple in r3 and every tuple in pt1. Subsequently, the same process would be done for the rest of r3's tuples.

It is clear that the resultant tuple, sent as a parameter to the insert_tuple method, will always have correct attributes to have values filled in; however, it will also have other attributes that will change from one invocation to the next. Additionally, the tuple to be entered into the resultant tuple will always be a constant type belonging to the class for which the particular insert_tuple implementation is specified. That is, the insert_tuple implementation for emp_tuple will always have a parameter tuple that is of type emp_tuple, with attributes person, address, phone, and widget, to be inserted into the resultant tuple. The resultant tuple parameter sent to this implementation of insert_tuple will always have attributes person, address, phone, and widget in addition to the attributes essn, proj_num, and hours.

Now the same operation is executed again except with pt1 replaced by a relation called R2 that has the attributes sponsor_ssn and dependent_name. An appropriate resultant tuple type, cart2_result_tuple, is created with the attributes person, address,

phone, widget, sponsor_ssn, and dependent_name. This time when insert_tuple is invoked for r3 it will receive two tuples: one of type emp_tuple and the other of type cart2_result_tuple. Again the resultant relation has the attributes that this implementation will requires, person, address, phone, and widget, but this time the resultant tuple also has the attributes sponsor_ssn and dependent_name. Thus, each implementation of insert_tuple must be able to handle differing structures for the resultant relation. This proved to be problematic for both the insert_tuple method and insert_field method.

The solution used in our R/OODBMS is to have two methods called insert_tuples and insert_fields. Instead of having every tuple type that belongs to some relation in the database, each resultant tuple type requires its own implementation. That is, in our example involving the Cartesian product operation, cart1_result_tuple and cart2_result_tuple would both have their own implementation of the method. There are now three parameters to the method: the two operand relations and the resultant relation. The entire concatenation process is completed within these methods and the resultant relation is returned from the method after completion. The method insert_fields works in a similar manner; however, it requires only two parameters, the operand relation and resultant relation.

c. Comparison methods

The comparison methods implemented in the R/OODBMS are equal_to, less_than, and greater_than. The equal_to method is used by the check union compatibility function and select operation. Although there are default implementations for these within the R/OODBMS, they have very limited applicability since the possible tuple types are limitless and each generally requires specific implementation for comparing tuples of the same type. Thus, the default implementations are a point of departure, but users must

provide their own implementations for each tuple type that is user defined if the R/ OODBMS is to perform properly.

The designer of the comparison operator methods has many decisions to make regarding their implementation. If the tuple type is complex with attributes that have a type that is a user defined object, then deciding at which level comparisons will be made may be difficult. For example, in Figure 25 the schema for an emp_tuple class is shown. An instance of this class has four attributes: person, address, phone, and widget. Each of these attributes is a user provided/defined class. Figure 32 shows the definition for these classes (except for widget[23]). Which level do the comparison operators compare at? When the person attribute of an emp_tuple is compared with another, are the object IDs (OIDs) of the objects that they reference compared? Or is every attribute of the person object: fname, mname, lname, ... compared? Or maybe the comparison should be made at a more detailed level?

```
person =>          fname     :  string,
                   mname     :  string,
                   lname     :  string,
                   bdate     :  string,
                   ssn       :  integer,
                   spouse    :  string,
                   sptr      :  person_nil;

addr =>            street    :  string,
                   city      :  string,
                   state     :  string,
                   zip       :  string;

phone_number =>    number    : string;
```

Figure 32 Person, Addr, Phone_number Class Definitions

23. The attribute widget is an arbitrarily complex attribute that has yet to be defined by the user.

As the attributes of a relation become more complex, it is easy to see how many variations there are for implementations of comparison operator methods. By the same token, it is also easy to see that one default implementation could never hope to be functional for more than just the simplest of attributes. Without user provided implementations for the comparison operator methods, user defined tuple types in R/OODBMS will not work properly.

3. User Definitions

The relations that can be formed within R/OODBMS are limited only by what the user can define within an IDL schema. However, it is critical that specific implementations for methods inherited from the IDB strict class tuple be written for all user defined tuple types. The implementation of the relational algebra operations for the IDB node class relation depend on them.

V. ALTERNATIVE PROJECT AND CARTESIAN PRODUCT IMPLEMENTATIONS

A. GENERAL

Of the five fundamental relational algebra operations (union, difference, select, Cartesian product, and project), Cartesian product and project are considered to be the most difficult. As discussed in Chapter IV, this is because the resultant relation yielded by these two operations have a different structure from the operand relation(s). For this reason, both ROOMS [Ne88] and our R/OODBMS require that the resultant relation structure for these two operations be defined by the user prior to the execution of the operation. This chapter deals with an IDB specific alternative solution to this approach.

Recall that an attribute that has a type that is of an IDB strict class type can reference any object that is a descendent of that strict class type. Thus, an attribute that is of type tuple can reference any object that is an instantiation of an IDB node class type which is a descendent of class tuple. In both ROOMS and R/OODBMS, the user is required to provide the definition of the resultant tuple. In Chapter IV, the only two resultant relation tuple types for Cartesian product and project operations that are defined are cart1_result_tuple and project1_result_tuple respectively. Hence, any result of either of these two operations (in the database as defined in the schema of APPENDIX C) must have the structure of cart1_result_tuple and project1_result_tuple. That is, the Cartesian product operation can only be executed on two relations where the first relation has tuples of type emp_tuple and the second relation has tuples of type proj_tuple. Similarly, the project operation can only project the hours and essn attributes of relations that have type proj_tuple.

Say that you wanted to perform a project operation on a relation with tuples of type proj_tuple for the attributes proj_num and hours. Or, that you wanted to take the Cartesian

product of a relation with proj_tuple types and another with emp_tuple types. In both cases, new tuple subclasses for the resultant relations would have to be defined prior to run-time. However, there is an alternative in IDB: the use of the type 'any'.

B. IDB TYPES

An attribute in an IDL schema can have any of the types listed in Figure 33. A R/OODBMS relation has an attribute called tuples that has type sequence of tuple where tuple is a IDB strict class type. Thus, any subclasses of tuple may be referenced by the class relation's attribute tuples. Therefore, a relation's schema is defined by the structure (tuple subclass definitions) of its tuples. In our R/OODBMS, the user must provide these subclass definitions to define their relations. In other words, the subclasses of class tuple are all user-defined classes. For example, in Chapter IV two tuple subclasses were used: emp_tuple and proj_tuple. Consider an Employee relation, it would have tuples that are of type emp_tuple. Our R/OODBMS relation's attribute tuples would then reference a sequence of emp_tuple.[1]

Another way to gain the same effect as having the relation's attribute tuples reference a different tuple subclass differing Cartesian product and project queries is to define one subclass that has the flexibility to dynamically reference varying number of tuple attributes and differing types of attributes with each instantiation. A tuple can be thought of as a sequence of attribute values, so if a linked list is used to allow the flexibility required in the sequence size then we simply need a type (this equates to the domain of an attribute) that can reference any object within the database. That is, each attribute in the resultant relation

1. It is important to note that emp_tuple and all other subclasses of tuple must be IDB node class types. That is, node class types may be instantiated into objects whereas strict class types cannot.

could have any domain that is already defined in the database. Any attribute that is of type 'any' can reference any object within the database.

<table>
<tr><td rowspan="6">reference</td><td colspan="2">string</td></tr>
<tr><td rowspan="2">sequence</td><td>array</td></tr>
<tr><td>linked</td></tr>
<tr><td rowspan="3">class</td><td>any</td></tr>
<tr><td>user-defined</td></tr>
<tr><td>nil</td></tr>
<tr><td rowspan="3">embedded</td><td rowspan="3">primitive</td><td>boolean</td></tr>
<tr><td>integer</td></tr>
<tr><td>rational</td></tr>
</table>

Figure 33 IDL Types[2] [Pe91c, p. 62]

C. THE RESULT_TUPLE SUBCLASS

In this alternative implementation, only one resultant relation tuple type is needed. It has been called result_tuple and has one attribute, values, which has type sequence of any (see Figure 34 and APPENDIX E[3]). Thus, this attribute can reference any object within the database. Since we cannot anticipate how many attributes any particular resultant relation will have, the sequence of any is a linked list of any which allows the number of attributes (values) within a result_tuple to vary from one result_tuple to another. Hence, this one tuple type can be used as the resultant tuple type for any operation in R/OODBMS.

---

2. The difference between reference and embedded types is important to note. Attributes that are reference types are pointers to an independent object that contains the attribute value. In contrast, attributes that are embedded types have their value imbedded within the object that it belongs.[Pe91c]

3. Changes from the original R/OODBMS schema (APPENDIX C) are in bold in APPENDIX E.

```
result_tuple        => values   :  seq of any;
```

Figure 34 Resultant Relation Schema for Project and Cartesian Product

Both the modified project operation and the modified Cartesian product operation create the resultant relation using a function that initializes all of the class relation attributes to initial values before the tuples are inserted into the relation. In both cases, the relation attribute tuples is initialized to reference an empty linked list of type result_tuple. A pointer to the new resultant relation is passed back to the calling function so that the tuples can be inserted into the relation. This initialization function differs for the two operations and will be discussed in further detail in the sections below. Each of these initialization functions (init_proj_result_rel and init_Cart_result_rel) is a variation of the init_temp_rel used by the simple operations (union, difference, and select).

D. THE MODIFIED OPERATIONS

1. Project

There are a number of functions used by the modified project operation that also required modification. These functions along with the modified project operation, project_op are listed in APPENDIX E. Note that these are only the functions from the original R/OODBMS shown in APPENDIX C. In order to have a working R/OODBMS with this alternative approach, the functions in APPENDIX E must be inserted into the code in APPENDIX C. Functions in the two appendices that have the same name indicate that the newer modified function replace the identically named function in APPENDIX C. All other functions in APPENDIX E that have unique names should simply be inserted.

The project query syntax is changed to that shown in Figure 35. The project_op function still implements the project operation. The difference between this implementation and the original begins with a modified function that parses the project query: Project_parse_action. There are fewer tokens to parse than in the original implementation.

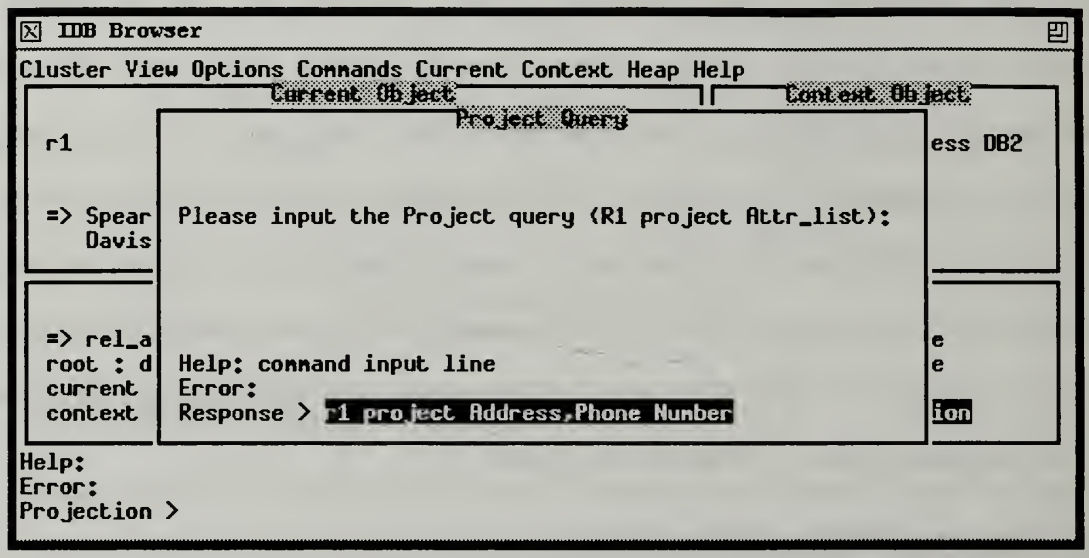


Figure 35 Modified Project Query

An additional data structure is required by this implementation since there is a need to pass an array of indexes as a parameter to the insert_fields_b function. This structure, index_array, is defined in the IDL schema in APPENDIX E; however, since it is not reachable from the root, it can never be written to the database.[4] Index_array has one attribute that is a sequence of integers that are used to indicate which attributes will be projected.

4. Remember that for an object to be written to secondary storage in IDB it must be reachable from the root.

Insert_fields_b is the workhorse of the function project_op. This modified function requires an additional parameter, index_array, that was not required by insert_fields. Otherwise, it still accepts two other parameters: the operand relation and resultant relation. The user must provide certain methods that override inherited methods during the construction of their relations. For example, the user must provide overriding implementation for the comparison operators for each relation. In the same vein, the user must provide an overriding implementation for the insert_fields_b method that they all inherit. The reason for this is that each relation has a different list of attributes that may be projected; thus one generic field (attribute) insertion method will not suffice. In the modified R/OODBMS schema only emp_tuple had an overriding implementation provided. Thus, project operations may only be done on relations with tuples that are of type emp_tuple type. The resultant relation from a project operation of a relation with proj_tuple types would yield meaningless results since no overriding implementation for insert_fields_b is provided in the schema.

Each user provided overriding implementation can be a copy of insert_emp_fields_b with a few modifications. Each attribute must have its own case that inserts that particular attribute into the resultant relation tuple (its index is in index_array). In Figure 36, insert_emp_fields_b has been modified to indicate the portions of the function that need to be changed for each relation within the database. Note that a new object is not created for each attribute that is inserted into the resultant relation; instead each attribute of the operand relation that is to be projected is referenced by the resultant relation tuple. This issue of creating an independent object for the resultant relation's attributes versus simply referencing the attributes of the operand relation(s) was discussed in Chapter IV and therefore it will not be discussed again here.

It should be noted that in this implementation the resultant relation from a project operation cannot itself be used as the operand relation for another project operation. This

could be done but would require a general overriding implementation of the insert_fields_b method that would not conform to the template shown in Figure 36. The fundamental difference between all user provided tuple subclass definitions and the result_tuple definition is that the user provided subclasses all have no attributes that are of a sequence types while the only attribute of the result_tuple is a sequence type. This difference will require that the attribute values are inserted into the resultant relation by use of idl_linked_for which will iterate through each of the attributes within the operand relation's tuples to find the appropriate attributes to insert. This is in contrast to using a case statement to insert the appropriate attributes.

The insert_fields_b method returns the resultant relation to project_op with all of the correct tuples inserted. Finally, the function project_op inserts the pointer to the new resultant relation into the sequence of relations that make up the database. Again, the resultant relation is not written to secondary storage until a transaction is opened that allows writing and is subsequently committed.

```
idl_routine relational_relation insert_XXXX_fields_b(rel,result_rel,index_array)
     relational_relation rel, result_rel;
     relational_index_array index_array;
{
  idl_transaction tr = idl_get_trans(rel);
  relational_result_tuple new_tuple;

  result_rel->tuples = idl_empty_linked(tr,relational_tuple);

  idl_linked_for (relational_tuple,rel->tuples,rel_tuple)
    {
      /* iterate through each tuple and for each tuple iterate through
         the index_array and use a case statement to reference objects for
         fields to be entered into the result relation */

     new_tuple = idl_new(tr,relational_result_tuple);
     new_tuple->values = idl_empty_linked(tr,relational_any);

     idl_linked_for (relational_index,index_array->indexes,index)
       {
         switch (index->i)
           {
           case 1:
             idl_insert_back(relational_any,new_tuple->values,rel_tuple->attr 1);
             break;
           case 2:
             idl_insert_back(relational_any,new_tuple->values,rel_tuple->attr 2);
             break;
           case 3:
             idl_insert_back(relational_any,new_tuple->values,rel_tuple->attr 3);
             break;
         •
         •
         •
           case n:
             idl_insert_back(relational_any,new_tuple->values,rel_tuple->attr n);
             break;

           default:
             idl_raise(IDL_ERROR,
                       "There is a problem in the employee insert field b
                                                                  function!");
             break;
           }
       } idl_end_for

     idl_insert_back(relational_tuple,result_rel->tuples,new_tuple);

   } idl_end_for
```

Figure 36 Template for Overriding Insert_fields_b Method

2. CARTESIAN PRODUCT

The discussion in the previous section regarding modified function's insertion into APPENDIX C also applies to those modified functions listed in APPENDIX F for Cartesian product. The Cartesian product query syntax is changed to that as shown in

Figure 37. The function cart_prod_op implements the Cartesian product operation. As with the project operation, the new query syntax dictates that there are fewer tokens to parse. As such, the function Cartesian_parse_action has been modified to parse this new query syntax.

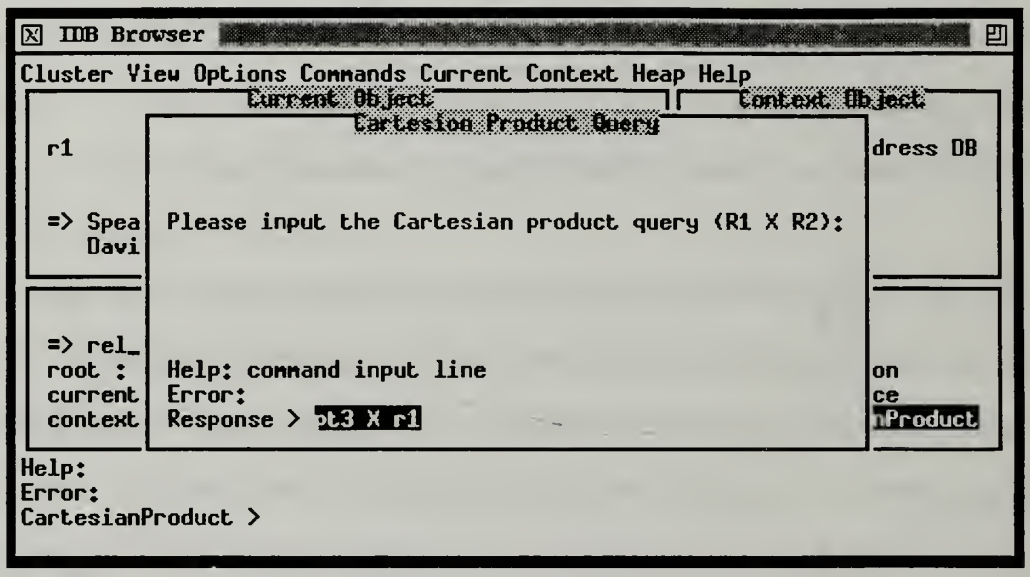


Figure 37 Modified Cartesian Product Query

The workhorse of the function cart_prod_op is the method insert_tuples_b. Insert_tuples_b still takes the same number and type of parameters as the original insert_tuples. In contrast to the insert_fields_b of the project operation, the insert_tuples_b method as defined for the class tuple does not require a user provided overriding implementation for each relation. Instead, the method insert_tuples_b is general enough that overriding implementations are not needed. The function does not need to worry about how many attributes each operand relation has. It simply iterates through every tuple in the first operand relation and concatenates the one tuple at a time with each tuple of the second operand relation (see code in APPENDIX F).

However, if the alternative approach to inserting tuples into the result relation is taken, then the user would have to provide an overriding implementation of insert_tuples_b for each relation as he must do for the insert_fields_b method of the project operation. The reason for this is that after an object has been created to hold the tuple to be entered into the resultant relation, the appropriate values must be explicitly copied into the newly created object. Once this has been done, then the new object with the tuples attribute values can be inserted into the resultant relation's tuple.

Since, with this implementation of insert_tuples_b, there is no need for any other redefinition of the method, the resultant relation yielded from a Cartesian product operation may be used as one of the operand relations in a subsequent Cartesian product query. As stated earlier, this is not possible with the project operation.

E. CONCLUSIONS

This alternative implementation of project and Cartesian product appears to be very promising. The project operation functions in the manner expected from a relational database perspective. That is, if you want to project attribute 3, attribute 5, and attribute 1 of a particular relation, then the resultant relation will have those three attributes only and they will be in that particular order. If you project attribute 5, attribute 3, and attribute 1, then the resultant relation will have three attributes that correspond to these three but in the order specified in the query. Similarly, Cartesian product yields the resultant relation that is expected.

There is one additional area that must be discussed, the displaying of the resultant relation. The approach taken in our R/OODBMS implementation was to provide a specific overriding implementation of the idl_print and idl_key methods for all relations. However, when the resultant relation is displayed in this implementation, the system defined implementations of idl_print and idl_key are used since a general overriding

implementation cannot be written for the resultant relation as each resultant relation may differ in structure. This makes it somewhat difficult to view the tuples of the resultant relation. However, this can easily be overcome as demonstrated in Nelson's implementation of ROOMS [Ne88] by having a display (overriding implementation of idl_print and idl_key in this implementation) implementation for every object in the database. That is, if an attribute of a relation is itself an object, then that object would have a specific implementation of the idl_print and idl_key methods.

VI. CONCLUSION

The purpose of this research was to implement a combined relational/object-oriented database management system, R/OODBMS, that will overcome the deficiencies/ constraints of separate relational and object-oriented systems. This thesis expands previous work in this area that showed basic proof of concept by implementing relational operations in an object-oriented programming language which did not provide for persistent objects [Ne88][NMO90], and by implementing relational operations in an object-oriented programming language which does provide for persistent objects [Fi92]. A R/OODBMS as implemented in IDB, a commercially-available object-oriented database management system, demonstrates the extension of previous work to a commercially-available object-oriented database management system.

A. SUMMARY

A detailed literature review investigating object-oriented programming concepts, relational database management systems, object-oriented database management systems (including a detailed overview of IDB), and a brief discussion of previous work in the area was accomplished. Limitations of conventional relational database management systems and object-oriented database management systems were explored while identifying desirable properties of a combined approach.

The R/OODBMS, as described in Chapter IV, is specific to an implementation of ROOMS [Ne88][NMO90] in IDB. However, as with previous work in this area, this research demonstrates the validity of the ROOMS concept and further completes and strengthens the proof of concept started by Nelson [Ne88].

B. CONCLUSIONS

A combination of relational and object-oriented database management systems, a R/OODBMS is a logical and viable solution in overcoming each individual system's limitations providing the best of both worlds within a single system. In a R/OODBMS, the relational paradigm gains the ability to model and manage arbitrarily complex data that traditional RDBMS are unable to handle. Additionally, the object-oriented paradigm gains the acceptance, standardization, and firm theoretical foundation that traditional RDBMSs enjoy. Thus, a single system can satisfy the requirements of users of both traditional systems and overcome many of their individual limitations.

C. FUTURE RESEARCH SUGGESTIONS

Future research in this area should generally be focused on the development of a production system. The methods used to implement the five fundamental relational algebra operations could be rewritten with efficiency and optimization in mind. During the implementation of the R/OODBMS in IDB as presented in this thesis, neither efficiency nor optimization were of great importance. Instead, the focus was on constructing an implementation that was functional.

In constructing queries, the user of our R/OODBMS does not have the option of naming the resultant relation for those operations (union, difference, and select) that create the resultant relation. The R/OODBMS creates a default name and assigns it to the resultant relation. A production system should allow the user the option to provide a name for the resultant relation if desired or to accept the default name provided by the system. For example, a query of the form 'R1 union R2' would indicate that the user is willing to accept the default name that the system provides. However, a query of the form 'R3 = R1 union R2' would indicate that the user would like the relations R1 and R2 to be unioned together and the result relation to be name R3.

There are several issues that will remain as design decisions for a production system. One issue involves how objects of the same class are compared with one another. Are two objects equal if they both reference the same OID? Or, should every attribute of the objects be compared to determine if each corresponding attribute has the same value for each object? A similar problem arises with the inequality comparison operators. When is one object greater than another? Even for complex objects that have attributes which are simple, this issue becomes difficult.

For example, consider an employee of a company that has a first name, middle name, last name, phone number, and street address. When the relation is defined within the R/OODBMS, the relation could be called Employee and have tuples of type employee_tuple. An employee_tuple might have three attributes defined: full_name, phone_number, and address where full_name and address are objects and phone_number is an integer (a local phone number without area code). The object full_name has attributes first_name, middle_name, and last_name where each attribute is a string. Address has attributes street, city, state, and zip_code where street, city and state are strings and zip_code is an integer. Now, consider the equal comparison operator. If we are trying to compare two employee tuples from two different relations, it is easy to determine if the phone_number in one tuple is equal to that of another. The value is simple an integer that may be compared directly. However, what if it is desirable to compare the full_name attribute? Realizing that it is itself an object, do we move to that object and consider each of its attributes (first_name, middle_name and last_name) as a whole? Do we compare OIDs? Or, do we allow a single attribute of full_name to be compared, such as only last_name? Or, do we allow any combination of the attributes to be compared for equality? This example can be generalized for any of the comparison operators whether they are equality or inequality comparison operators.

The issue of comparison operator implementation is also of concern in the set operations union and difference during the determination of union compatibility and the determination of whether a tuple is a duplicate of another tuple within a relation. This issue is also present in the select operation where attributes of different tuples are compared during as part of the selection criteria.

The assumption was made during this implementation of a R/OODBMS that the user made no errors. Therefore, limited error checking was built into the system. A production system would require extensive error checking for all of the relational algebra operations.

Another issue is found in the insertion of tuples into resultant relations. Our R/OODBMS uses two methods of insertion: by reference or by creation. That is, the tuples of the resultant relation may have new tuples created and then the values that the tuple is to have are copied into the attributes of the tuple. Alternatively, the tuples of a resultant relation can simply be a reference to the tuple that contains the values to be inserted.

In this second approach, two (or more) different relations can refer to the same tuple. Thus, a change to the tuple in either relation is reflected in the other relation. This approach is problematic since if there is a duplicate tuple in two relations that are being unioned together, the resultant relation will only reference one of the two. Now, a change to the tuple in the resultant relation will be reflected in only one of the two operand relations. This is not very consistent and requires that the user be aware of how the system determines which tuple is going to be referenced in the resultant relation. If the first approach is used, the problem is alleviated since every tuple in the resultant relation is independent of every tuple in any of the operand relations.

Finally, the project operation in a production system should allow the user to rename the attributes being projected into the resultant relation. However, the issue of attributes that are complex objects comes to light again. In our employee example, if a project operation were used to project all the names of employees and to change the name of the

attribute in the resultant relation, is the user only allowed to change the attribute full_name to some other name? Or, is the user allowed to change each of the attribute names (such as first_name, middle_name, and last_name)? Or, is the user allowed to change any one or more of these attribute names?

Additionally, our R/OODBMS requires the user to form a select query using the syntax 'R1 select <attribute name> <comparison operator> <comparison object>' where the comparison object has the same superclass as R1 and has only a single tuple. A production system should also allow the user to construct a select query of the syntax 'R1 select <attribute name> <comparison operator> <value>' where value is the display form <attribute name>. For example, in our employee example, the user should be allowed to enter the query 'R1 select phone_number = 3758619'.

APPENDIX A: EXAMPLE IDL SCHEMA

```
structure vhc root dict is
   dict =>           local          : fleet,
                     remote         : cross of fleet;
      fleet =>       vehicles       : seq of vehicle;
         vehicle ::= train | plane;
         vehicle =>  location       : city,
                     destination    : city_nil;
         vehicle ->  move(*,city,city);

   city =>           name : string;
   city_nil ::=      city | nil;

   train ::=      ptrain | ftrain;
   train =>       cars               : seq of car;
      car =>      number             : integer;
      ptrain =>;  ftrain =>;

   plane ::=      pplane | fplane;
   plane =>       airborne           : boolean;
      pplane =>;  fplane =>;
   passenger ::=  ptrain | pplane;
   passenger =>   max                : integer,
                  passengers         : seq of person;
   freight ::=    ftrain | fplane;
   freight =>     cargo_weight       : rational;
   person =>      name               : string;

   for fleet.vehicles use linked;
   for train.cars use linked;
   for passenger.max use unsigned;
   for passenger.max use bytes(2);

   for vehicle.idl_print use bind(vehicle_print);
   for train.idl_print use bind(train_print);
   for vehicle.move use bind(vehicle_move);

   for vehicle.location use search;
   for vehicle.location use search_embed;
   for ptrain use description("passenger train");
   for ftrain use description("freight train");
   for pplane use description("plane (passenger)");
   for fplane use description("air freight");
   for passenger.max use
      description("maximum number of passengers");
end
process vhcp is vhc ::= vhca : access; end
```

APPENDIX B: DATABASE DIRECTORY SOURCE CODE

```
--*****************************************************************************
--
-- dbdir.idl
--
--   Schema for database directory
--
-- Description:
--
--    A directory cluster is used to group together other clusters into
--    a directory. In this directory, the clusters are databases.
--
--*****************************************************************************

structure dbdir root seq of database is

for root use linked;

database =>  dbname  : string,            -- database name
             file    : string,            -- database file name
             desc    : string,            -- description of database cluster
             examine : boolean;           -- default initial transaction

for database.dbname use description("database structure name");
for database.file use description("database cluster file name");
for database.desc use description("description of the database cluster");
for database.examine use description("enter in examine mode?");
for database.desc use search;

database -> enter(*,boolean) => boolean;

for database.enter use browser_cond_visible;
for database.enter use description("enter this cluster");

for database.idl_key use bind(dbdir_key);
for database.idl_print use bind(dbdir_print);
for database.idl_create use bind(dbdir_icreate);
for database.enter use bind(dbdir_enter);

end

process dbdirp is

   dbdir ::= dbdira:access;

end
```

```
/*****************************************************************************
*
* dbdir.c
*
*
* Description:
*
*   This module provides the operations for IDB database directories.
*   This module is considered to be part of the IDB browser.  It invokes
*   internal browser routines to enter and exit database clusters.
*
*****************************************************************************/

#include "stdio.h"
#include "idlrt.h"
#include "dbdira.h"
#include "dpy.h"
#include "brwa.h"
#include "brw.h"
#include "brwi.h"

static char buff[100];


/*****************************************************************************
*
*   OPERATIONS
*
*   _brw_dbdir_key(*)                        -- print short form db description
*   _brw_dbdir_print(*,mode)                 -- print full display
*   _brw_dbdir_icreate(tr) => *              -- create database object
*   _brw_dbdir_enter(*,boolean) => boolean   -- enter cluster
*
*****************************************************************************/


idl_routine void _brw_dbdir_key(db)
     dbdir_database db;
{
  dpy_cstring(db->desc);
}
```

```
idl_routine void _brw_dbdir_print(db,mode)
     dbdir_database db;
     dpy_dmode mode;
{
  if (mode.expand > 0)
    {
      idl_top(idl_any,idl_print,(db,mode));
      return;
    }
  dpy_open("",false);
  dpy_attr(dbdir_database,db,desc,mode);
  if (idl_get_display(dbdir_database,dbname))
    {
      dpy_eol();
      dpy_cstring("type:");
      dpy_attr(dbdir_database,db,dbname,mode);
    }
  if (idl_get_display(dbdir_database,file))
    {
      dpy_eol();
      dpy_cstring("file:");
      dpy_attr(dbdir_database,db,file,mode);
    }
   if (idl_get_display(dbdir_database,examine))
     {
       dpy_eol();
       dpy_cstring("examine:");
       dpy_attr(dbdir_database,db,examine,mode);
     }
  dpy_close();
}
```

```
static dbdir_database new_db;

idl_routine void _brw_dbdir_daction(check)
     integer check;
{
  _idl_not_used(dummy);
  if (check == 1)
    {
      integer fd;
      string full_name = _brw_map_db(new_db->dbname);
      if (idl_string_size(new_db->dbname) == 0)
        {
          dpy_error("no database name specified");
          return;
        }
      (void) sprintf(buff,"%s.bst",full_name);
      if (fd > 0)
        {
          (void) close(fd);
        }
      else
        {
          (void) sprintf(buff,"unable to find symbol table file %s.bst",
                         full_name);
          dpy_error(buff);
          return;
        }
      if (idl_string_size(new_db->file) == 0)
        {
          dpy_error("no structure name specified");
          return;
        }
      if (idl_string_size(new_db->desc) == 0)
        {
          new_db->desc = new_db->file;
          dpy_error("defaulting description");
          return;
        }
    }
  dpy_quit();
}

idl_routine void _brw_dbdir_sscreen(a,x,y)
     integer a,x,y;
{
  dpy_dmode mode;

  _idl_not_used(a);
  mode = dpy_dmode_default;

  dpy_open("Define New Database",true);

  dpy_open("New Database",true);
  brw_cmd("Check","check and exit if correct",_brw_dbdir_daction,1L,BRW_SCREEN);
  dpy_spacex(5L);
  brw_cmd("Nocheck","exit without checking",_brw_dbdir_daction,0L,BRW_SCREEN);
  dpy_eol();
  dpy_spacey(2L);
  dpy_eol();
  mode.expand = 1;
  idl_vop(new_db,dbdir_database,idl_print,(new_db,mode));
  dpy_close();

  brw_input_area(y-3,false);

  dpy_close();
  dpy_boxed(x,y);
}
```

```
idl_routine dbdir_database _brw_dbdir_icreate(tr)
     idl_transaction tr;
{
  string empty = idl_copy_string(tr,"");
  new_db = idl_new(tr,dbdir_database);
  new_db->dbname = empty;
  new_db->file = empty;
  new_db->desc = empty;
  new_db->examine = true;
  dpy_active(_brw_dbdir_sscreen,0L);
  return new_db;
}



idl_routine boolean _brw_dbdir_enter(db,test)
     dbdir_database db;
     boolean test;
{
  if (! test)
    {
      integer tkind = BRW_READ;
      idl_if (brw_tr,_brw_dict->curr_tr,btr)
        {
          (void) sprintf(buff,"%s/%s",btr->name,db->desc);
        }
      idl_else
        {
          (void). sprintf(buff,"**unknown**/%s",db->desc);
        } idl_end_if

      /* open new transaction */
      if (db->examine) tkind = BRW_EXAMINE;
      (void) _brw_topen(idl_copy_string(brw_static,buff),
                        db->dbname,db->file,
                        0L,
                        BRW_NORMAL,tkind,true);
    }
  return true;
}


idl_define_ops dbdir_opbind()
{
  idl_bind_root(dbdir);
  idl_bind("dbdir_key",_brw_dbdir_key);
  idl_bind("dbdir_print",_brw_dbdir_print);
  idl_bind("dbdir_icreate",_brw_dbdir_icreate);
  idl_bind("dbdir_enter",_brw_dbdir_enter);
}
```

APPENDIX C: R/OODBMS SOURCE CODE

Contents

```
////////////////////////////////////////////////////////////////////////////////
```

IDL SCHEMA

```
////////////////////////////////////////////////////////////////////////////////

--============================================================================
--Title       : A Relational/Object-Oriented Database Management System
--File name   : relational.idl
--Associated
--  files     : relational.c relationala.h relational.bst relational.so
--Author      : Ronald L. Spear
--Date        : 24 September 1992
--            : Master's Thesis
--Advisor     : Maj Mike Nelson
--Second
--  Reader    : Prof. Thomas Wu
--System      : Sun 4/60 Workstation, Unix Operating System
--Compiler    : AT&T C
--Translator  : IDL v1.1, Persistent Data Systems
--Description : This file contains the IDL schema for the implementation of
--              A Relational/Object-Oriented Database Management System.
--============================================================================

structure relational root database is

--*****************************************************************************

////////////////////////////////////////////////////////////////////////////////
                                Database Schema
////////////////////////////////////////////////////////////////////////////////


   database =>    name              : string,
                  relations         : seq of relation;

   database ->    new_relation(*);

////////////////////////////////////////////////////////////////////////////////
                                Relation Schema
////////////////////////////////////////////////////////////////////////////////

   relation =>    relation_name     : string,
                  attribute_names   : seq of name,
                  attribute_types   : seq of name,
                  tuples            : seq of tuple,
                  tuple_type        : tuple,
                  key               : string;

   relation ->    new_tuple(*),
                  check_union_compatability(relation,relation) => boolean;

   relation ->    union(*),
                  projection(*),
                  difference(*),
                  Cartesian_product(*),
                  selection(*);

      name =>     name              : string;
```

```
////////////////////////////////////////////////////////////////////////////////
                                Tuple Schema
////////////////////////////////////////////////////////////////////////////////

   tuple ->       equal_to(tuple,tuple,integer)              => boolean,
                  less_than(tuple,tuple,integer)             => boolean,
                  greater_than(tuple,tuple,integer)          => boolean,

                  initialize_tuple(tuple)                    => tuple,
                  insert_fields(relation,relation)           => relation,
                  insert_tuples(relation,relation,relation)  => relation;

////////////////////////////////////////////////////////////////////////////////
                              Tuple Subclasses
////////////////////////////////////////////////////////////////////////////////

   tuple ::=      emp_tuple |
                  proj_tuple |
                  cart1_result_tuple |
                  project1_result_tuple |
                  nil;

--*****************************************************************************

   for database.new_relation use browser_visible;

   for relation.new_tuple use browser_visible;
   for relation.union use browser_visible;
   for relation.Cartesian_product use browser_visible;
   for relation.difference use browser_visible;
   for relation.projection use browser_visible;
   for relation.selection use browser_visible;


   for database.relations use linked;
   for relation.tuples use linked;

--******************** Database, Relation, and Tuple Methods *********************

   for database.idl_key use bind(database_key);
   for database.idl_print use bind(database_print);
   for database.new_relation use bind(create_relation);

   for relation.idl_key use bind(relation_key);
   for relation.idl_print use bind(relation_print);
   for relation.new_tuple use bind(create_tuple);
   for relation.check_union_compatability use bind(ck_union_compatability);

   for relation.union use bind(union_op);
   for relation.Cartesian_product use bind(cart_prod_op);
   for relation.difference use bind(set_diff_op);
   for relation.projection use bind(project_op);
   for relation.selection use bind(select_op);

   for name.idl_key use bind(name_key);
   for name.idl_print use bind(name_print);

   for tuple.equal_to use bind(equal_to);
   for tuple.less_than use bind(less_than);
   for tuple.greater_than use bind(greater_than);

   for tuple.initialize_tuple use bind(initialize_tuple);
   for tuple.insert_fields use bind(insert_fields);
   for tuple.insert_tuples use bind(insert_tuples);
```

```
--**************************************emp_tuple*********************************

////////////////////////////////////////////////////////////////////////////////
                              Employee Tuple Schema
////////////////////////////////////////////////////////////////////////////////

   emp_tuple =>          person    :  person,
                         address   :  addr,
                         phone     :  phone_number,
                         widget    :  idl_univ;

      person =>          fname     :  string,
                         mname     :  string,
                         lname     :  string,
                         bdate     :  string,
                         ssn       :  integer,
                         spouse    :  string,
                         sptr      :  person_nil;

         person_nil ::= person | nil;

      addr =>            street    :  string,
                         city      :  string,
                         state     :  string,
                         zip       :  string;

      phone_number =>    number    :  string;
--**********************************emp_tuple methods**************************

   for emp_tuple.idl_key use bind(emp_tuple_key);
   for emp_tuple.idl_print use bind(emp_tuple_print);
   for emp_tuple.equal_to use bind(emp_equal_to);
   for emp_tuple.less_than use bind(emp_less_than);
   for emp_tuple.greater_than use bind(emp_greater_than);
   for emp_tuple.initialize_tuple use bind(initialize_emp_tuple);

   for person.idl_key use bind(person_key);
   for person.idl_print use bind(person_print);

   for addr.idl_key use bind(addr_key);
   for addr.idl_print use bind(addr_print);

   for phone_number.idl_print use bind(phone_number_print);
```

```
--************************************proj_tuple*********************************

////////////////////////////////////////////////////////////////////////////////
                              Project Tuple Schema
////////////////////////////////////////////////////////////////////////////////

   proj_tuple =>          essn     : integer,
                          proj_num : integer,
                          hours    : rational;

--*********************************proj_tuple mehods*****************************

   for proj_tuple.idl_key use bind(proj_tuple_key);
   for proj_tuple.idl_print use bind(proj_tuple_print);
   for proj_tuple.equal_to use bind(proj_equal_to);
   for proj_tuple.less_than use bind(proj_less_than);
   for proj_tuple.greater_than use bind(proj_greater_than);
   for proj_tuple.initialize_tuple use bind(initialize_proj_tuple);

--*********************************cart1_result_tuple****************************

////////////////////////////////////////////////////////////////////////////////
                          Cartesian1 Result Tuple Schema
////////////////////////////////////////////////////////////////////////////////

   cart1_result_tuple =>  person  :  person,
                          address  :  addr,
                          phone    :  phone_number,
                          widget   :  idl_univ,
                          essn     :  integer,
                          proj_num :  integer,
                          hours    :  rational;

--*****************************cart1_result_tuple methods************************

   for cart1_result_tuple.idl_key use bind(cart1_result_tuple_key);
   for cart1_result_tuple.idl_print use bind(cart1_result_tuple_print);
   for cart1_result_tuple.initialize_tuple use
               bind(initialize_cart1_resul_tuple);
   for cart1_result_tuple.insert_tuples use bind(insert_cart1_result_tuples);

--*******************************************************************************

////////////////////////////////////////////////////////////////////////////////
                          Project1 Result Tuple Schema
////////////////////////////////////////////////////////////////////////////////

   project1_result_tuple   => hours :  rational,
                              essn  :  integer;

--***************************project1_result_tuple methods**********************

   for project1_result_tuple.idl_key use bind(project1_result_tuple_key);
   for project1_result_tuple.idl_print use bind(project1_result_tuple_print);
   for project1_result_tuple.initialize_tuple use
               bind(initialize_project1_resul_tuple);
  for project1_result_tuple.insert_fields use bind(insert_project1_result_flds);

--*******************************************************************************


end
```

```
--*****************************************************************************
process relationalp is
   relational ::= relationala:access;
end
```

```
/*
 * ============================================================================
 * Title       : A Relational/Object-Oriented Database Management System
 * File name   : relational.c
 * Author      : Ronald L. Spear
 * Date        : 24 September 1992
 *             : Master's Thesis
 * Advisor     : Maj Mike Nelson
 * Second
 *   Reader    : Prof. Thomas Wu
 * System      : Sun 4/60 Workstation, Unix Operating System
 * Compiler    : AT&T C
 * Translator  : IDL v1.1, Persistent Data Systems
 * Description : The functions in this file run with the IDB Object Database
 *               system version 1.1. The file containing the schema for these
 *               methods is relaitonal.idl. The relational.idl file was
 *               translated using the IDL translator and produced the
 *               relationala.h header file which is included for use with
 *               this file.
 * ============================================================================
 */

#include <string.h>
#include "idlrt.h"
#include "relationala.h"
#include "dpy.h"
#include "brw.h"

/*****************************************************************************
*
*    Macros defined for Class person
*
*****************************************************************************/

#define exists(v) idl_string_size(v) != 0

#define rexists(node,attr) (idl_get_display(relational_person,attr) && \
            (node->attr != 0 && \
            exists(node->attr)))

#define dexists(node,attr) (mode.expand > 0 || rexists(node,attr))

#define prexists(node,attr) (idl_get_display(relational_person,attr) && \
            (node->attr != 0))

#define pdexists(node,attr) (mode.expand > 0 ||\
             prexists(node,attr))
```

```
/*****************************************************************************
*
*    Forward References for all functions other than _print and _key
*
*****************************************************************************/

idl_routine void init_rel_screen();
idl_routine void create_relation();
idl_routine void init_action();
idl_routine void init_tuple_screen();
idl_routine void create_tuple();
idl_routine boolean ck_union_compatability();
idl_routine void exit_action();
idl_routine void char_screen();
idl_routine void integer_screen();
static void union_parse_action();
idl_routine relational_relation init_temp_rel();
void report_union_error();
static void difference_parse_action();
void report_difference_error();
static void Cartesian_parse_action();
void report_Cart_product_error();
static void Project_parse_action();
void report_project_error();
static void Select_parse_action();
void report_select_error();
idl_routine void union_op();
idl_routine void cart_prod_op();
idl_routine void set_diff_op();
idl_routine void project_op();
idl_routine void select_op();
idl_routine boolean equal_to();
idl_routine boolean less_than();
idl_routine boolean greater_than();
idl_routine boolean emp_equal_to();
idl_routine boolean emp_less_than();
idl_routine boolean emp_greater_than();
idl_routine boolean proj_equal_to();
idl_routine boolean proj_less_than();
idl_routine boolean proj_greater_than();
idl_routine relational_relation insert_fields();
idl_routine relational_relation insert_tuples();
idl_routine relational_tuple initialize_emp_tuple();
idl_routine relational_tuple initialize_tuple();
idl_routine relational_tuple initialize_proj_tuple();
idl_routine relational_tuple initialize_cart1_resul_tuple();
idl_routine relational_relation insert_cart1_result_tuples();
idl_routine relational_tuple initialize_project1_resul_tuple();
idl_routine relational_relation insert_project1_result_flds();
```

```
/*****************************************************************************
*
*    Class database methods
*
*        database_key
*        database_print
*        create_relation
*
*****************************************************************************/
*****************************************************************************

                              Class Database Methods
*****************************************************************************

/* displays a short description of a database object */

idl_routine void database_key(database)
     relational_database database;
{
  idl_univ u;
  if (idl_string_size(database->name) == 0)
    {
      dpy_cstring("** Unnamed Database **");
    }
  else
    {
      u = idl_to(idl_univ,database->name);
      idl_vop(u,idl_univ,idl_key,(u));
    }
}

/*===========================================================================*/
```

```
/* diplays a database object */

idl_routine void database_print(database,mode)
     relational_database database;
     dpy_dmode mode;
{
  idl_transaction tr = idl_get_trans(database);
  boolean can_write = idl_trans_write_count(tr) > 0;
  dpy_dmode mode1;

  mode1 = mode;
  mode1.embed = 1;

  if (can_write)
    {
      mode1.expand = 1;
      mode.expand =1;
    }

  if (mode1.expand > 1)
    {
      idl_top(idl_any,idl_print,(database,mode));
    }

  dpy_attr(relational_database,database,name,mode);
  dpy_eol();
  dpy_spacey(2L);
  dpy_eol();
  dpy_attr(relational_database,database,relations,mode1);
  dpy_eol();
}

/*=============================================================================*/
```

```
static relational_relation init_relation; /* global ptr to new rel to
                                             be initialized */

/* this function provides the user a pop-up screen in the browser from which
   he may edit/initialize the values of a new relation. It is called by
   create_relation. */

idl_routine void init_rel_screen(a,x,y)
   integer a,x,y;
{
   dpy_dmode mode;

   mode = dpy_dmode_default;
   dpy_open("Initialize New Relation",true);
   dpy_open("New Relation",true);
   brw_cmd("EXIT","exit initialization screen",init_action,0L,BRW_SCREEN);
   dpy_eol();
   dpy_spacey(2L);
   dpy_eol();
   mode.expand = 1; /* forces the display of all attributes */

   /* execute the print method for the database class/type */
   idl_vop(init_relation,relational_database,idl_print,(init_relation,mode));
   dpy_close();

   brw_input_area(y-3,false);

   dpy_close();
   dpy_boxed(x,y);
}
```

```
/* Create Relation creates a new relation within a database.*/

idl_routine void create_relation(database)
     relational_database database;
{
  relational_relation new_relation;
  idl_transaction tr = idl_get_trans(database);
  string empty = idl_copy_string(tr,"");
  boolean is_writable = (idl_trans_write_count(tr) > 0);

  new_relation = idl_new(tr,relational_relation); /* must still assign
                                                     legal values,which is
                                                     done below. */

  new_relation->relation_name = empty;

  /* this is only done to make attribute names valid. An array with a different
     size can be created while in the browser and new_relation->attribute_names
     can be switched to reference it if a larger array is needed. */

  new_relation->attribute_names = idl_new_array(tr,relational_name,1);
  new_relation->attribute_names[0] = idl_new(tr,relational_name);
  new_relation->attribute_names[0]->name = empty;

  new_relation->attribute_types = idl_new_array(tr,relational_name,0);

  new_relation->tuples = idl_empty_linked(tr,relational_tuple);

  new_relation->tuple_type = NULL;

  new_relation->key = empty;

  /* set global pointer to new relation so that the attributes of the new
     relation may be initialized by the user. Note: that dpy_active does
     not allow other parameters to be passed, thus a global pointer is used
     so that init_rel_screen can access the new relation */

  init_relation = new_relation;
  dpy_active(init_rel_screen,0L);

  /* if a write transaction is open, then add the new relation to the database */
  if (is_writable)
    idl_insert_back(relational_relation,database->relations,new_relation);
}
```

```
/*****************************************************************************
*
*    Class relation methods
*
*        relation_key
*        relation_print
*        create_tuple
*        ck_union_compatability
*
*        union_op
*        cart_prod_op
*        set_diff_op
*        project_op
*        select_op
*
*****************************************************************************/
*****************************************************************************

                              Class Relation Methods
*****************************************************************************


/* Displays a short description of a relation */

idl_routine void relation_key(relation)
     relational_relation relation;
{
  idl_univ u;
  if (idl_string_size(relation->relation_name) == 0)
    {
      dpy_cstring("** Unnamed Relation **");
    }
  else
    {
      u = idl_to(idl_univ,relation->relation_name);
      idl_vop(u,idl_univ,idl_key,(u));
    }
}

/*===========================================================================*/
```

```
/* Displays a relation */

idl_routine void relation_print(relation,mode)
     relational_relation relation;
     dpy_dmode mode;
{
  idl_transaction tr = idl_get_trans(relation);
  boolean can_write = idl_trans_write_count(tr) > 0;
  dpy_dmode mode1;

  mode1 = mode;
  mode1.embed = 1;

  if (can_write)
    {
      mode1.expand = 1;
      mode.expand = 1;
    }

  if (mode1.expand > 1)
    {
      idl_top(idl_any,idl_print,(relation,mode));
    }

  dpy_attr(relational_relation,relation,relation_name,mode);
  dpy_eol();
  dpy_spacey(2L);
  dpy_eol();
  dpy_attr(relational_relation,relation,tuples,mode1);
  dpy_eol();
  if (can_write) /* Don't want to display all of these attributes unless
                    there is a transaction open which allows writing to the
                    database. This information is needed for the implementation
                    of the R/OODBMS but is not needed by the user. However,
                    these attributes must be assigned appropriate values if
                    the relational algebra operations of the system are to
                    function properly. */
    {
      dpy_spacey(2L);
      dpy_eol();
      dpy_attr(relational_relation,relation,attribute_names,mode1);
      dpy_eol();
      dpy_spacey(2L);
      dpy_eol();
      dpy_attr(relational_relation,relation,attribute_types,mode1);
      dpy_eol();
      dpy_spacey(2L);
      dpy_eol();
      dpy_attr(relational_relation,relation,tuple_type,mode1);
      dpy_eol();
      dpy_spacey(2L);
      dpy_eol();
      dpy_attr(relational_relation,relation,key,mode1);
    }
}

/*============================================================================*/
```

```
static relational_tuple init_tuple; /* ptr to new tuple to be initialized */

/* Init action allows init_xxxxx_action functions to exit if EXIT is selected
   in the pop up screen for initialization */

idl_routine void init_action(val)
   boolean val;
{
   dpy_quit();
}

/* init_tuple_screen called by create_tuple. It allows the user to initialize
   the attributes of a new tuple. */

idl_routine void init_tuple_screen(a,x,y)
   integer a,x,y;
{
   dpy_dmode mode;

   mode = dpy_dmode_default;
   dpy_open("Initialize New Tuple",true);
   dpy_open("New Tuple",true);
   brw_cmd("EXIT","exit initialization screen",init_action,0L,BRW_SCREEN);
   dpy_eol();
   dpy_spacey(2L);
   dpy_eol();
   mode.expand = 1; /* forces all attributes to be displayed */

   /* executes print method for relation class/type */

   idl_vop(init_tuple,relational_relation,idl_print,(init_tuple,mode));
   dpy_close();

   brw_input_area(y-3,false);

   dpy_close();
   dpy_boxed(x,y);
}
```

```
/* create_tuple creates and inserts a new tuple into a relation. It calls
   tuple method   initialize_tuple for the tuple type of the current relation.
   Each subclass  of tuple must have a redefinition of the class tuple's
   method initialize_tuple. If there is no redefinition, then the function
   initialize_tuple is  called and causes an IDL error to be raised. */

idl_routine void create_tuple(relation)
     relational_relation relation;
{
  relational_tuple new_tuple;
  idl_transaction tr = idl_get_trans(relation);
  string empty = idl_copy_string(tr,"");
  boolean is_writable = (idl_trans_write_count(tr) > 0);

  new_tuple = idl_vop(relation->tuple_type,relational_tuple,initialize_tuple,
                      (relation->tuple_type));

  /* uses global ptr to new tuple so it can be initialized in function
     init_tuple_screen */

  init_tuple = new_tuple;
  dpy_active(init_tuple_screen,0L);

  /* check to see if a write transaction is open before inserting the new
     tuple into the relation. If there is not one open, then the new tuple
     is lost */
  if (is_writable)
    idl_insert_back(relational_tuple,relation->tuples,new_tuple);
  }

/*=============================================================================*/
```

```
/* ck_union_compatability takes two relations and determines if they are union
   compatible. This means that the two relations have the same number
   of attributes and that the corresponding attribute types are the same -
   in the same order. That is, they are the of the same order and
   R1_attr(i) = R2_attr(i) for 1 less than or equal to i and i less
   than or equal to n. */

idl_routine boolean ck_union_compatability(ptr_R1,ptr_R2)
     relational_relation ptr_R1,ptr_R2;
{
  boolean same_order,types_equal;
  idl_transaction tr;
  integer R1_degree,R2_degree,i,R1_types,R2_types;

  tr = idl_get_trans(ptr_R1);
  types_equal = true;

  R1_degree = idl_array_size(ptr_R1->attribute_names);
  R2_degree = idl_array_size(ptr_R2->attribute_names);
  R1_types = idl_array_size(ptr_R1->attribute_types);
  R2_types = idl_array_size(ptr_R2->attribute_types);

  /* check to make sure attribute types have legal values */
  if ((R1_types == 0) && (R2_types == 0))
    idl_raise(IDL_ERROR,
              "Both R1 and R2 have illegal values for their attribute_types
attributes!");
  else
    {
      if (R1_types == 0)
        idl_raise(IDL_ERROR,
                  "R1 has an illegal value for its attribute_types attribute!");

      if (R2_types == 0)
        idl_raise(IDL_ERROR,
                  "R2 has an illegal value for its attribute_types attribute!");
    }

  /* check to make sure attribute names have legal values */
  if ((R1_degree == 0) && (R2_degree == 0))
    idl_raise(IDL_ERROR,
              "Both R1 and R2 have illegal values \nfor their attribute_names
attributes!");
  else
    {
      if (R1_degree == 0)
        idl_raise(IDL_ERROR,
                  "R1 has an illegal value for \nits attribute_names attribute!");

      if (R2_degree == 0)
        idl_raise(IDL_ERROR,
                  "R2 has an illegal value for \nits attribute_names attribute!");
    }

  /* check the order of each relation */
  same_order = (R1_degree == R2_degree);

  /* each attribute must be of some type. Thus, there must be the same number of
     of elements in the attribute_names array as there are in the attribute_types
     array. */

  if ((R1_degree != R1_types) && (R2_degree != R2_types))
    idl_raise(IDL_ERROR,
              "Neither R1 and R2 have the a one-to-one correspondence\nbetween the
number of attributes and types listed in their\nattribute_names and
attribute_types attibutes!");

  else
```

```
    {
      if (R1_degree != R1_types)
        idl_raise(IDL_ERROR,
                  "R1 does not have a one-to-one correspondence\nbetween its
nattribute_names and attribute_types attibutes!");

      if (R2_degree != R2_types)
        idl_raise(IDL_ERROR,
                  "R2 does not have a one-to-one correspondence\nbetween its
nattribute_names and attribute_types attibutes!");
    }

  if (same_order)
    {
      /* check the corresponding types of each */
      for (i = 0; i < R2_degree; ++i)
        {
          types_equal = ((strcmp(ptr_R1->attribute_types[i]->name,
                                 ptr_R2->attribute_types[i]->name)) == 0);
          if (!types_equal)
            break;
        }
    }

  if (same_order && types_equal)
    {
      return true; /* the two relations are union compatable */
    }
  else
    {
      if (!same_order)
        idl_raise(IDL_ERROR,
                  "The two relations are not union compatible!\nThey are not of
the same order.");
      else
        idl_raise(IDL_ERROR,
                  "The two relations are not union compatible!\nThey do not have
equal corresponding types.");

      return false;
    }
)

/*============================================================================*/
```

```
char *R1,*R2,*R3; /* global ptrs to the parameters for the union operation.
                     R1 union R2 */

/* union_parse_action is called by the brw_input operation within the union_op
   function. Union parse action takes the query string and parses it into the
   two operand relations R1 and R2. */

static void union_parse_action(query)
     char* query;
{
  char *R1_ptr,*char_ptr;
  integer size,i;
  boolean done = false;

  char_ptr = query;

  /* allocate room for parse of the union op parameters
     R1 will hold the first parameter and R2 the second */
  size = strlen(query);
  R1 = (char*)calloc((size+1),sizeof(char));
  R2 = (char*)calloc((size+1),sizeof(char));

  R1_ptr = R1;
  char_ptr = query;

  /* note: if size gets decremented all the way to zero, then there is a
     problem with the query because the delimeter ' union ' could not be
     found */
  while (!done && size > 0)
    {
      if (*char_ptr != ' ') /* if not a space copy the char into R1 */
        {
          *R1_ptr=*char_ptr;
          ++char_ptr;
          ++R1_ptr;
          --size;
        }
      else /* we may have hit the delimeter for the first parameter */
        {
          /* check to see if next char is a "u" - the first letter of union
             which is the delimeter between the two parameters */
          if (strncmp(char_ptr," union ",7) == 0) /* then it is union sentinal */
            {
              for (i = 0; i < 7;++i) /* jump past the delimeter */
                {
                  ++char_ptr;
                  --size;
                }
              strcpy(R2,char_ptr); /* copy second parameter into R2 */
              done=true;
            }
          else /* the space is part of the first parameter, so put in R1 */
            {
              /* space is part of first relation name so keep it */
              *R1_ptr=*char_ptr;
              ++char_ptr;
              ++R1_ptr;
              --size;
            }
        }
    }

  if (size != 0) /* size only = 0 if union was not found in the query */
    NULL;
  else
    idl_raise(IDL_ERROR,
              "There is an error in your union query! Try Again.");
}
```

```
/* init_temp_rel is called by union_op, set_diff_op and select_op since they
   are the three simple relational algebra operations that have a resultant
   relation that is of the same structure as the operand relations. Init temp
   rel creates a new relation that is to be the resultant relation of one of
   the three listed operations. It assigns default values and then passes a
   reference to the new relation to the caller. */

idl_routine relational_relation init_temp_rel(ptr_R1,ptr_R2)
     relational_relation ptr_R1,ptr_R2;
{
  relational_relation temp_relation;
  static integer temp_rel_num = 0;
  idl_transaction tr;
  string temp_rel_name;
  char temp_name[80];
  integer degree = 0,i;

  tr = idl_get_trans(ptr_R1);

  degree = idl_array_size(ptr_R1->attribute_names);

  temp_relation = idl_new(tr,relational_relation); /* must still assign
                       legal values */

  /* set up a unique name for resultant relation */
  sprintf(temp_name,"%1dTEMP_%c%c.%c%c",
          ++temp_rel_num,
          ptr_R1->relation_name[0],
          ptr_R1->relation_name[1],
          ptr_R2->relation_name[0],
          ptr_R2->relation_name[1]);
  temp_rel_name = idl_copy_string(tr,temp_name);

  temp_relation->relation_name = temp_rel_name;
  temp_relation->attribute_names = idl_new_array(tr,relational_name,degree);

  /* assign default values for attribute names to be the same as those in
     R1 relation */
  for ( i=0; i<degree; ++i)
    {
      temp_relation->attribute_names[i] = ptr_R1->attribute_names[i];
    }

  temp_relation->attribute_types = ptr_R1->attribute_types;

  temp_relation->tuples = idl_empty_linked(tr,relational_tuple);

  /* assign a default tuple type that is the same as the first relations */
  temp_relation->tuple_type = ptr_R1->tuple_type;

  /* default key is the key of relation R1 */
  temp_relation->key = ptr_R1->key;

  return temp_relation;
}
```

```
/* report_union_error reports errors as the name implies. It was taken out
   of union_op and made into a separate function to make union_op more
   readable. This has also been done with the other four operations in
   the R/OODBMS. */

void report_union_error(found1,found2)
     boolean found1,found2;
{
  if (!found1 && !found2)
    {
      idl_raise(IDL_ERROR,
                "Neither relation is in this database!");
    }
  else
    {
      if (!found1)
        {
          idl_raise(IDL_ERROR,
                    "R1 (the first parameter) is not in this database!");
        }
      if (!found2)
        {
          idl_raise(IDL_ERROR,
                    "R2 (the second parameter) is not in this database!");
        }
    }
  if (found1 && found2)
    {
      idl_raise(IDL_ERROR,
                "A SERIOUS ERROR HAS OCCURED IN THE UNION OPERATION!");
    }
}
```

```
/* union_op is executed when Union is selected within the browser. The
   syntax for the operation allows the user to input the two relations
   to be unioned and then creates the resultant relation (with an assigned
   default name). It calls union_parse_action, init_temp_rel,
   check_union_compatability and report_union_error. */
*******************************************************************************
```

Union Method

```
*******************************************************************************

idl_routine void union_op(relation)
     relational_relation relation;
(
  relational_relation ptr_R1,ptr_R2,temp_relation;
  relational_database database;
  idl_trans_mode tmode;
  idl_univ root;
  string parameter1,parameter2;  /* references to the parameters R1 and R2 */
  static integer temp_rel_num = 0,index = 0;
  idl_transaction tr;
  boolean found1,found2,is_writable = false,duplicate = true,compatible = false;

  tr = idl_get_trans(relation);
  tmode = idl_trans_mode_default;
  root = idl_trans_get_root(tr);
  database = idl_to(relational_database,root);
  found1 = false;
  found2 = false;
  is_writable = (idl_trans_write_count(tr) > 0);

  brw_input("Union Query",
            "Please input the union query (R1 union R2): ",
            0L,0L,0L,false,
            union_parse_action);

  /* copy the C strings R1 and R2 into IDL strings */
  parameter1 = idl_copy_string(tr,R1);
  parameter2 = idl_copy_string(tr,R2);

  /* search the database for the two relations: R1 and R2 */
  idl_linked_for (relational_relation,database->relations,rel)
    {
      if (strcmp (rel->relation_name,parameter1) == 0) /* found relation 1 */
        {
          ptr_R1 = rel; /* point at relation 1 */
          found1 = true;
        }

      if (strcmp (rel->relation_name,parameter2) == 0) /* found relation 2 */
        {
          ptr_R2 = rel;
          found2 = true;
        }
    ) idl_end_for

  if (found1 && found2)
    {
      /* check for union compatability */
      compatible = idl_vop(ptr_R1,relational_relation,check_union_compatability,
                           (ptr_R1,ptr_R2));

      if (compatible)
        (
          temp_relation = init_temp_rel(ptr_R1,ptr_R2);

          /* insert the tuples from R1 and R2 into Temp relation. First
```

```
            all the tuples from R1 are inserted into Temp relation. */

          idl_linked_for (relational_tuple,ptr_R1->tuples,tuple)
            {
              idl_insert_back(relational_tuple,temp_relation->tuples,tuple);
            } idl_end_for

          /* don't insert any duplicate tuples into the resultant relation.
             Thus, have to check the tuple key with each tuple already in the
             result (or in R1) before entering another tuple into the
             resultant relation */

          idl_linked_for (relational_tuple,ptr_R2->tuples,r2_tuple)
            {
              idl_linked_for (relational_tuple,ptr_R1->tuples,r1_tuple)
                {

                  /* duplicate check takes the two tuples and
                     determines if they have the same attribute values. If yes,
                     then they are duplicates. It returns a boolean. */

                  duplicate = idl_vop(r1_tuple,relational_tuple,equal_to,
                                      (r1_tuple,r2_tuple,index));

                  if (duplicate)
                    {
                      break;
                      /* get out of loop cuz found tuple was a duplicate */
                    }
                } idl_end_for

              if (!duplicate)
                {
                  idl_insert_back(relational_tuple,
                                  temp_relation->tuples,r2_tuple);
                }
            } idl_end_for

          /* If a write transaction is open when this command is executed then
             if it becomes committed the temp relation will be written to the
             database. However, if a read or examine transaction is open, then
             the temp relation will be in the database in memory but will not
             be written back to secondary storage. */

          idl_insert_back(relational_relation,database->relations,
                          temp_relation);
        }
      else /* else part of if (compatible)*/
        {
          idl_raise(IDL_ERROR,
                    "The two relations are not union compatible!");
        }
    }
  else /* else part of if (found1 && found2) */
    {
      report_union_error(found1,found2);
    }
}

/*=============================================================================*/
```

```
/* difference_parse_action is called by the brw_input operation within the
   set_diff_op function. Difference parse action takes the query string and
   parses it into the two operand relations R1 and R2. */

static void difference_parse_action(query)
     char* query;
{
  char *R1_ptr,*char_ptr;
  integer size,i;
  boolean done = false;

  char_ptr = query;

  /* allocate room for parse of the union op parameters
     R1 will hold the first parameter and R2 the second */
  size = strlen(query);
  R1 = (char*)calloc((size+1),sizeof(char));
  R2 = (char*)calloc((size+1),sizeof(char));

  R1_ptr = R1;

  /* do the parse */
  char_ptr = query;

  /* note: if size gets decremented all the way to zero, then there is a
     problem with the query because the delimeter ' union ' could not be
     found */
  while (!done && size > 0)
    {
      if (*char_ptr != ' ') /* if not a space copy the char into R1 */
        {
          *R1_ptr=*char_ptr;
          ++char_ptr;
          ++R1_ptr;
          --size;
        }
      else /* we may have hit the delimeter for the first parameter */
        {
          /* check to see if next char is a "-" which is the delimeter between
             the two parameters */
         if (strncmp(char_ptr," - ",3) == 0) /* then it is difference sentinal */
            {
              for (i = 0; i < 3;++i) /* jump past the delimeter */
                {
                  ++char_ptr;
                  --size;
                }
              strcpy(R2,char_ptr); /* copy second parameter into R2 */
              done=true;
            }
          else /* the space is part of the first parameter, so put in R1 */
            {
              /* space is part of first relation name so keep it */
              *R1_ptr=*char_ptr;
              ++char_ptr;
              ++R1_ptr;
              --size;
            }
        }
    }

  if (size != 0) /* size only = 0 if union was not found in the query */
    NULL;
  else
    idl_raise(IDL_ERROR,
              "There is an error in your difference query! Try Again.");
}
```

```
/* report_difference_error reports errors as the name implies. It was taken out
   of set_diff_op and made into a separate function to make set_diff_op more
   readable. This has also been done with the other four operations in
   the R/OODBMS. */

void report_difference_error(found1,found2)
     boolean found1,found2;
{
  if (!found1 && !found2)
    {
      idl_raise(IDL_ERROR,
                "Neither relation is in this database!");
    }
  else
    {
      if (!found1)
        {
          idl_raise(IDL_ERROR,
                    "R1 (the first parameter) is not in this database!");
        }
      if (!found2)
        {
          idl_raise(IDL_ERROR,
                    "R2 (the second parameter) is not in this database!");
        }
    }
  if (found1 && found2)
    {
      idl_raise(IDL_ERROR,
                "A SERIOUS ERROR HAS OCCURED IN THE DIFFERENCE OPERATION!");
    }
}
```

```
/* set_diff_op is executed when Difference is selected within the browser.
   The syntax for the operation allows the user to input the two relations
   to be operated on and then creates the resultant relation. It calls
   difference_parse_action, init_temp_rel, check_union_compatability
   and report_difference_error. */
*******************************************************************************
```

Difference Method

```
*******************************************************************************

idl_routine void set_diff_op(relation)
     relational_relation relation;
{
  relational_relation ptr_R1,ptr_R2,temp_relation;
  relational_database database;
  idl_trans_mode tmode;
  idl_univ root;
  string parameter1,parameter2;  /* references to the parameters R1 and R2 */
  static integer temp_rel_num = 0,index = 0;
  idl_transaction tr;
  boolean found1,found2,is_writable = false,duplicate = true,compatible = false;

  tr = idl_get_trans(relation);
  tmode = idl_trans_mode_default;
  root = idl_trans_get_root(tr);
  database = idl_to(relational_database,root);
  found1 = false;
  found2 = false;
  is_writable = (idl_trans_write_count(tr) > 0);

  brw_input("Difference Query",
            "Please input the Difference query (R1 - R2): ",
            0L,0L,0L,false,
            difference_parse_action);

  /* copy the C strings R1 and R2 into IDL strings */
  parameter1 = idl_copy_string(tr,R1);
  parameter2 = idl_copy_string(tr,R2);

  /* search the database for the two relations: R1 and R2 */
  idl_linked_for (relational_relation,database->relations,rel)
    {
      if (strcmp (rel->relation_name,parameter1) == 0) /* found relation 1 */
        {
          ptr_R1 = rel; /* point at relation 1 */
          found1 = true;
        }

      if (strcmp (rel->relation_name,parameter2) == 0) /* found relation 2 */
        {
          ptr_R2 = rel;
          found2 = true;
        }
    } idl_end_for

  if (found1 && found2)
    {
      /* check for union compatability */
      compatible = idl_vop(ptr_R1,relational_relation,check_union_compatability,
                           (ptr_R1,ptr_R2));

      if (compatible)
        {
          temp_relation = init_temp_rel(ptr_R1,ptr_R2);
```

```
        /* don't insert any tuples from R1 that are in R2 into the
           resultant relation. Thus, have to check the tuple key with each
           tuple in R2 before entering a tuple into the resultant relation */

        idl_linked_for (relational_tuple,ptr_R1->tuples,r1_tuple)
          {
            idl_linked_for (relational_tuple,ptr_R2->tuples,r2_tuple)
              {

                /* equal_to takes the two tuples and determines if
                   they have the same attribute values. If yes,
                   then they are equal. It returns a boolean. */

                duplicate = idl_vop(r1_tuple,relational_tuple,equal_to,
                                    (r1_tuple,r2_tuple,index));

                if (duplicate) /* then don't insert tuple into result */
                  {
                    break;
                    /* get out of loop cuz found tuple was a duplicate */
                  }
              } idl_end_for

            if (!duplicate)
              {
                idl_insert_back(relational_tuple,
                                temp_relation->tuples,r1_tuple);
              }
          ) idl_end_for


        /* If a write transaction is open when this command is executed then
           if it becomes committed the temp relation will be written to the
           database. However, if a read or examine transaction is open, then
           the temp relation will be in the database in memory but will not
           be written back to secondary storage. */

        idl_insert_back(relational_relation,database->relations,
                        temp_relation);
      }
    else /* else part of if (compatible)*/
      {
        idl_raise(IDL_ERROR,
                  "The two relations are not union compatible!");
      }
  }
 else /* else part of if (found1 && found2) */
  {
    report_difference_error(found1,found2);
  }
}

/*=============================================================================*/
```

```
/* Cartesian_parse_action is called by the brw_input operation within the
   cart_prod_op function. Cartesian parse action takes the query string and
   parses it into the two operand relations R1 and R2 along with the name
   of the resultant relation, R3. */

static void Cartesian_parse_action(query)
     char* query;
{
  char *R1_ptr,*char_ptr,*R3_ptr;
  integer size,i;
  boolean done = false,delimeter1 = false;

  char_ptr = query;

  /* allocate room for parse of the Cartesian product op parameters
     R1 will hold the first parameter and R2 the second */
  size = strlen(query);
  R1 = (char*)calloc((size+1),sizeof(char));
  R2 = (char*)calloc((size+1),sizeof(char));
  R3 = (char*)calloc((size+1),sizeof(char));

  /* set pointers to move along R1 and R3 as characters are copied in one
     at a time. */
  R3_ptr = R3;
  R1_ptr = R1;

  /* do the parse */
  char_ptr = query;

  /* note: if size gets decremented all the way to zero, then there is a
     problem with the query because the delimeter ' = ' or ' X ' could not be
     found */
  while (!done && size > 0)
    {
      if (*char_ptr != ' ') /* if not a space copy the char into R3 */
        {
          if (!delimeter1)
            {
              *R3_ptr=*char_ptr;
              ++char_ptr;
              ++R3_ptr;
              --size;
            }
          else
            {
              *R1_ptr=*char_ptr;
              ++char_ptr;
              ++R1_ptr;
              --size;
            }
        }
      else /* we may have hit the a delimeter */
        {
          if (!delimeter1)
            {
              /* check to see if next char is a "=" - which separates the
                 resultant relation name from the other relations in the query -
                 which is the delimeter between the first two parameters */
              if (strncmp(char_ptr," = ",3) == 0) /* then it is = sentinal */
                {
                  delimeter1 = true;
                  for (i = 0; i < 3;++i) /* jump past the delimeter */
                    {
                      ++char_ptr;
                      --size;
                    }
                }
              else /* the space is part of the first parameter, so put in R1 */
```

```
                {
                  /* space is part of first relation name so keep it */
                  *R3_ptr=*char_ptr;
                  ++char_ptr;
                  ++R3_ptr;
                  --size;
                }
            }
          else
            {
              /* check to see if next char is a " X " - which separates the
                 two operands of the operation - which is the delimeter
                 between the last two parameters */

              if (strncmp(char_ptr," X ",3) == 0) /* then it is X sentinal */
                {
                  for (i = 0; i < 3;++i) /* jump past the delimeter */
                    {
                      ++char_ptr;
                      --size;
                    }
                  strcpy(R2,char_ptr); /* copy second parameter into R2 */
                  done=true;
                }
              else /* the space is part of the first parameter, so put in R1 */
                {
                  /* space is part of first relation name so keep it */
                  *R1_ptr=*char_ptr;
                  ++char_ptr;
                  ++R1_ptr;
                  --size;
                }
            }
        }
    }

  if (size != 0) /* size only = 0 if union was not found in the query */
    NULL;
  else
    idl_raise(IDL_ERROR,"There is an error in your query! Try Again.");
}
```

```
/* report_Cart_product_error reports errors. It was taken out
   of cart_prod_op and made into a separate function to make cart_prod_op more
   readable. This has also been done with the other four operations in
   the R/OODBMS. */

void report_Cart_product_error(found1,found2,found3)
     boolean found1,found2,found3;
{
  if (!found1 && !found2 && !found3)
    {
      idl_raise(IDL_ERROR,
                "None of the three relations are in this database!");
    }
  else
    {
      if (!found1)
        {
          idl_raise(IDL_ERROR,
                    "R1 is not in this database!");
        }
      if (!found2)
        {
          idl_raise(IDL_ERROR,
                    "R2 is not in this database!");
        }
      if (!found3)
        {
          idl_raise(IDL_ERROR,
                    "R3 is not in this database!");
        }
      if (found1 && found2 && found3)
        {
          idl_raise(IDL_ERROR,
                    "A SERIOUS ERROR HAS OCCURED !!!!!! Regroup. Try Again.");
        }
    }
}
```

```
/* cart_prod_op is executed when CartesianProduct is selected within the
   browser. The syntax for the operation allows the user to input the two
   relations to be operated on along with the name of the resultant relation.
   It calls Cartesian_parse_action, report_Cart_product_error, and
   insert_tuples. */
*******************************************************************************
```

Cartesian Product Method

```
*******************************************************************************


idl_routine void cart_prod_op(relation)
     relational_relation relation;
{
  relational_relation ptr_R1,ptr_R2,ptr_R3,temp_relation;
  relational_database database;
  idl_trans_mode tmode;
  idl_univ root;
  string parameter1,parameter2,result_rel;  /* references to the parameters
                                               R1, R2 and R3 respectively */
  idl_transaction tr;
  boolean found1,found2,found3;
  boolean is_writable = false,duplicate = true,compatible = false;

  tr = idl_get_trans(relation);
  tmode = idl_trans_mode_default;
  root = idl_trans_get_root(tr);
  database = idl_to(relational_database,root);
  found1 = false;
  found2 = false;
  found3 = false;
  is_writable = (idl_trans_write_count(tr) > 0);

  brw_input("Cartesion Product Query",
            "Please input the Cartesian product query (R3 = R1 X R2): ",
            0L,0L,0L,false,
            Cartesian_parse_action);

  /* copy the C strings R1 and R2 into IDL strings */
  parameter1 = idl_copy_string(tr,R1);
  parameter2 = idl_copy_string(tr,R2);
  result_rel = idl_copy_string(tr,R3);

  /* don't do anything if the resultant relation is one of the two operands.
     However, the resultant relation can be one that exists in the data.
     In this case, the specified resultant relation will be over written. */

  if (!(strcmp (result_rel,parameter1)==0) &&
      !(strcmp (result_rel,parameter2)==0))
    {
      /* search the database for the three relations: R1, R2 and R3 */
      idl_linked_for (relational_relation,database->relations,rel)
        {
          if (strcmp (rel->relation_name,parameter1) == 0)
            /* found relation 1 */
            {
              ptr_R1 = rel; /* point at relation 1 */
              found1 = true;
            }

          if (strcmp (rel->relation_name,parameter2) == 0)
            /* found relation 2 */
            {
              ptr_R2 = rel;
              found2 = true;
            }
```

```
          if (strcmp (rel->relation_name,result_rel) == 0)
            /* found relation 3 */
            {
              ptr_R3 = rel;
              found3 = true;
            }
        } idl_end_for

      if (found1 && found2 && found3)
        {
          /* perform concatenation of tuples for Cartesian product.
             Note, in this implementation, the resultant relation already
             exists in the database. Thus, there is no need to insert any
             new relations into the database. We only have to fill in the
             resultant relation structure that already exists. */

          ptr_R3 = idl_vop(ptr_R3->tuple_type,relational_tuple,insert_tuples,
                           (ptr_R1,ptr_R2,ptr_R3));

        }
      else
        {
          report_Cart_product_error(found1,found2,found3);
        }

    } /* if result relation is one of operands */
  else
    {
      idl_raise(IDL_ERROR,
              "R3, the resultant relation is one of the two operand relations.\nIt
must be a relation in the database but not\none of the two operands!");
    }
}

/*============================================================================*/
```

```
char *Attr_list; /* allows global access to the attribute list for the project
                    operation */

/* Project_parse_action is called by the brw_input operation within the
   project_op function. Project parse action takes the query string and
   parses it into the single operand relation R1, the result relation R2
   along with the attribute list that is to be projected. */

static void Project_parse_action(query)
     char* query;
{
  char *R1_ptr,*char_ptr,*R2_ptr;
  integer size,i;
  boolean done = false,delimeter1 = false;

  char_ptr = query;

  /* allocate room for parse of the project op parameters
     R1 will hold the relation being operated on,
     Attr_list the list of attr to be projected, and
     R2 the resultant relation  */
  size = strlen(query);
  R1 = (char*)calloc((size+1),sizeof(char)); /* R1 is global */
  Attr_list = (char*)calloc((size+1),sizeof(char));
  R2 = (char*)calloc((size+1),sizeof(char)); /* R2 is global */

  /* set pointers to move along R1 and R2 as characters are copied in one
     at a time. */
  R2_ptr = R2;
  R1_ptr = R1;

  /* do the parse */
  char_ptr = query;

  /* note: if size gets decremented all the way to zero, then there is a
     problem with the query because the delimeter ' = ' or ' project '
     could not be found */
  while (!done && size > 0)
    {
      if (*char_ptr != ' ') /* if not a space copy the char into R2 */
        {
          if (!delimeter1)
            {
              *R2_ptr=*char_ptr;
              ++char_ptr;
              ++R2_ptr;
              --size;
            }
          else
            {
              *R1_ptr=*char_ptr;
              ++char_ptr;
              ++R1_ptr;
              --size;
            }
        }
      else /* we may have hit a delimeter */
        {
          if (!delimeter1)
            {
              /* check to see if next char is a "=" - which separates the
                 resultant relation name from the other relation in the query -
                 which is the delimeter between the first two parameters */
              if (strncmp(char_ptr," = ",3) == 0) /* then it is = sentinal */
                {
                  delimeter1 = true;
                  for (i = 0; i < 3;++i) /* jump past the delimeter */
                    {
```

```
                    ++char_ptr;
                    --size;
                  }
              }
            else /* the space is part of the first parameter, the resultant
                    relation name, so put in R2 */
              {
                *R2_ptr=*char_ptr;
                ++char_ptr;
                ++R2_ptr;
                --size;
              }
          }
        else /* we have already found the first delimeter */
          {
            /* check to see if next char is a " p " - which is
               part of the delimeter " project "between the last
               two parameters */

            if (strncmp(char_ptr," project ",9) == 0)
              /* then it is project */
              {
                for (i = 0; i < 9;++i) /* jump past the delimeter */
                  {
                    ++char_ptr;
                    --size;
                  }
                strcpy(Attr_list,char_ptr);
                /* copy list of attributes into Attr_list and now parse the
                   list of attributes */

                done=true;
              }
            else /* the space is part of the first parameter, so put in R1 */
              {
                /* space is part of first relation name so keep it */
                *R1_ptr=*char_ptr;
                ++char_ptr;
                ++R1_ptr;
                --size;
              }
          }
      }
  }

  if (size != 0) /* size only = 0 if union was not found in the query */
    NULL;
  else
    idl_raise(IDL_ERROR,"There is an error in your query! Try Again.");
}
```

```
/* report_project_error reports errors. It was taken out
   of project_op and made into a separate function to make project_op more
   readable. This has also been done with the other four operations in
   the R/OODBMS. */

void report_project_error(found1,found2,attr_found)
     boolean found1, found2, attr_found;
{
  if (!found1 && !found2)
    {
      idl_raise(IDL_ERROR,
                "Neither of the two relations are in this database!");
    )
  else
    {
      if (!attr_found)
        {
          idl_raise(IDL_ERROR,
                  "All of the attributes in the attribute list \nare not in R1!");
        }
      if (!found1)
        {
          idl_raise(IDL_ERROR,
                    "R1 is not in this database!");
        }
      if (!found2)
        {
          idl_raise(IDL_ERROR,
                    "R2 is not in this database!");
        }

      if (found1 && found2)
        {
          idl_raise(IDL_ERROR,
                    "A SERIOUS ERROR HAS OCCURED !!!!!! Regroup. Try Again.");
        }
    }
}
```

```
/* project_op is executed when Projection is selected within the
   browser. The syntax for the operation allows the user to input the single
   relation to be operated on along with the name of the resultant relation and
   the attribute list to be projected. It calls Project_parse_action,
   report_project_error, and insert_fields. */
*******************************************************************************
```

Project Method

```
*******************************************************************************


idl_routine void project_op(relation)
     relational_relation relation;
{
  relational_relation ptr_R1,ptr_Attr_list,ptr_R2,temp_relation;
  relational_database database;
  idl_trans_mode tmode;
  idl_univ root;
  string parameter1,attr_string,result_rel;  /* references to the parameters
                                                R1, attr list and result
                                                relation  respectively */
  idl_transaction tr;
  boolean found1,found2,done,attr_found;
  boolean is_writable = false,duplicate = true,compatible = false;
  char *attr_ptr,*delimeter = ",";  /* delimeter between elements
                                       in attribute list */

  string *attr_list[100];
  integer i=1,count,size,index,index_array[100],ii=0;
  idl_linked_elem(relational_tuple) result_tuple;

  tr = idl_get_trans(relation);
  tmode = idl_trans_mode_default;
  root = idl_trans_get_root(tr);
  database = idl_to(relational_database,root);
  found1 = false;
  found2 = false;
  done = false;
  is_writable = (idl_trans_write_count(tr) > 0);


  brw_input("Project Query",
            "Please input the Project query (R2 = R1 project Attr_list): ",
            0L,0L,0L,false,
            Project_parse_action);


  /* copy the C strings R1, R2 and Attr_list into IDL strings */
  parameter1 = idl_copy_string(tr,R1);
  result_rel = idl_copy_string(tr,R2);
  attr_string = idl_copy_string(tr,Attr_list);

  /* parse tokens in attribute string */
  if ((attr_ptr = strtok(attr_string,delimeter)) == NULL)
    {
      /* error, no token */

      idl_raise(IDL_ERROR,
             "You did not list any attribute/field names in\nyour project query!
Try again, meathead!");
    }
  else
    {
      attr_list[0] = idl_new_string(tr,80);
      attr_list[0] = idl_copy_string(tr,attr_ptr);
    }
```

```
while ((attr_ptr = strtok(NULL,delimeter)) != NULL)
  {
    attr_list[i] = idl_new_string(tr,80);
    attr_list[i] = idl_copy_string(tr,attr_ptr);
    ++i;
  }

/* don't do anything if the resultant relation is the operand relation.
   However, the resultant relation can be one that exists in the data.
   In this case, the specified resultant relation will be over written. */

if (!(strcmp (result_rel,parameter1)==0))
  {
    /* search the database for the two relations: R1 and R2 */
    idl_linked_for (relational_relation,database->relations,rel)
      {
        if (strcmp (rel->relation_name,parameter1) == 0)
          /* found relation 1 */
          {
            ptr_R1 = rel; /* point at relation 1 */
            found1 = true;
          }

        if (strcmp (rel->relation_name,result_rel) == 0)
          /* found relation 2 */
          {
            ptr_R2 = rel;
            found2 = true;
          }
      } idl_end_for

    count = i; /* count is the number of tokens - 1 */

    /* check each attr name in the attr list of the project operation to
       ensure that the field exists in the relation R1 */

    for(i=0;i<count;++i)
      {
        attr_found = false;
        ii=0;

        idl_array_for(relational_name,ptr_R1->attribute_names,aname)
          {
            ii++; /* position in attribute list */
            if (strcmp (aname->name,attr_list[i]) == 0) /* attr name in attr
                                                           list is a field
                                                           of R1 */
              {
                attr_found = true;
                index_array[i]=ii;
                break;
              }
          } idl_end_for

        if (!attr_found)
          {
            break; /* an attr in the project attr list is not
                      in the relation R1. Thus, the operation
                      cannot be performed */
          }
      }

    if (found1 && found2 && attr_found)
      {
        /* everything is ok, perform projection operation on relation R1.
           Note, in this implementation, the resultant relation already
           exists in the database. Thus, there is no need to insert any
           new relations into the database. We only have to fill in the
```

```
            resultant relation structure that already exists. */

          /* iterate through every tuple of R1 and copying only the desired
             fields into R2. Note: R1 and R2 will have the same number of
             tuples but the relations will be of differing degree. */


          ptr_R2 = idl_vop(ptr_R2->tuple_type,relational_tuple,insert_fields,
                           (ptr_R1,ptr_R2));

        }
      else
        {
          report_project_error(found1,found2,attr_found);
        }
    }
  else /* end of if (!(strcmp (result_rel,parameter1)==0)) */
    {
      idl_raise(IDL_ERROR,
                "R2, the resultant relation is also the operand.\nIt must be a
relation in the database but not the operand!");
    }

}

/*============================================================================*/
```

```
char *Attr,*Obj,*Comp_opr; /* allows global access to three stings: an attribute
                              name used for the select condition, the name of
                              the object created that contains the select values
                            for comparison, and the comparison operator name. */

/* Select_parse_action is called by the brw_input operation within the
   select_op function. Select parse action takes the query string and
   parses it into the single operand relation R1, the attribute that is
   going to be used to select upon, the comparison operator to be used,
   and the name of the object that contains the attribute value to be
   compared with.*/

static void Select_parse_action(query)
     char* query;
{
  char *R1_ptr,*char_ptr,*Attr_ptr,*Comp_opr_ptr;
  integer size,i;
  boolean done = false,delimeter1 = false;

  char_ptr = query;

  /* allocate room for parse of the select op parameters
     R1 will hold the relation being operated on,
     Attr the attr that selection will be made on,
     Comp_opr the comparison operator (=,<,>), and
     Obj holds the values to be compared with */

  size = strlen(query);
  R1 = (char*)calloc((size+1),sizeof(char)); /* R1 is global */
  Attr = (char*)calloc((size+1),sizeof(char)); /* Attr is global */
  Obj = (char*)calloc((size+1),sizeof(char)); /* Obj is global */
  Comp_opr = (char*)calloc((size+1),sizeof(char)); /* Comparison Operator
                                                      is global */

  /* set pointers to move along R1 and Obj as characters are copied in one
     at a time. */
  Attr_ptr = Attr;
  R1_ptr = R1;
  Comp_opr_ptr = Comp_opr;

  /* do the parse */
  char_ptr = query;

  /* note: if size gets decremented all the way to zero, then there is a
     problem with the query because the delimeter ' select '
     could not be found */
  while (!done && size > 0)
    {
      if (*char_ptr != ' ') /* if not a space copy the char into R1 */
        {
          if (!delimeter1)
            {
              *R1_ptr=*char_ptr;
              ++char_ptr;
              ++R1_ptr;
              --size;
            }
          else
            {
              *Attr_ptr=*char_ptr;
              ++char_ptr;
              ++Attr_ptr;
              --size;
            }
        }
      else /* we may have hit a delimeter */
        {
          if (!delimeter1)
```

```
  {
    /* check to see if next char is a "select" -
       which is the delimeter between the first two parameters */
    if (strncmp(char_ptr," select ",8) == 0) /* then it is select */
      {
        delimeter1 = true;
        for (i = 0; i < 8;++i) /* jump past the delimeter */
          {
            ++char_ptr;
            --size;
          )
      }
    else /* the space is part of the first parameter, the operand
            relation name, so put in R1 */
      {
        *R1_ptr=*char_ptr;
        ++char_ptr;
        ++R1_ptr;
        --size;
      }
  }
else /* we have already found the first delimeter */
  {
    /* check to see if the next token is a comparison operator */

    if (strncmp(char_ptr," = ",3) == 0 ||
        strncmp(char_ptr," < ",3) == 0 ||
        strncmp(char_ptr," > ",3) == 0)
      {
        ++char_ptr;
        --size;
        for (i = 0; i < 1;++i) /* copy comparison operator */
          {
            *Comp_opr_ptr=*char_ptr;
            ++Comp_opr_ptr;
            ++char_ptr;
            --size;
          }
        ++char_ptr;
        --size;
        strcpy(Obj,char_ptr);
        done=true;
      }
    else if (strncmp(char_ptr," /= ",4) == 0 ||
             strncmp(char_ptr," <= ",4) == 0 ||
             strncmp(char_ptr," >= ",4) == 0)
      {
        ++char_ptr;
        --size;
        for (i = 0; i < 2;++i) /* copy comparison operator */
          {
            *Comp_opr_ptr=*char_ptr;
            ++Comp_opr_ptr;
            ++char_ptr;
            --size;
          )
        ++char_ptr;
        --size;
        strcpy(Obj,char_ptr);
        done=true;
      )
    else /* the space is part of the first parameter, so put in R1 */
      {
        /* space is part of first relation name so keep it */
        *Attr_ptr=*char_ptr;
        ++char_ptr;
        ++Attr_ptr;
        --size;
```

```
                    }
                }
            }
        }

    if (size != 0) /* size only = 0 if union was not found in the query */
        NULL;
    else
        idl_raise(IDL_ERROR,"There is an error in your query! Try Again.");

}
```

```
/* report_select_error reports errors. It was taken out
   of select_op and made into a separate function to make select_op more
   readable. This has also been done with the other four operations in
   the R/OODBMS. */

void report_select_error(found1,found2,attr_found)
     boolean found1, found2, attr_found;
{
  if (!found1 && !found2)
    {
      idl_raise(IDL_ERROR,
                "Neither of the two relations are in this database!");
    }
  else
    {
      if (!attr_found)
        {
          idl_raise(IDL_ERROR,
                    "The attribute specified for comparison is not in R1!");
        }
      if (!found1)
        {
          idl_raise(IDL_ERROR,
                    "R1 is not in this database!");
        }
      if (!found2)
        {
          idl_raise(IDL_ERROR,
                    "The object for comparison is not in this database!");
        }

      if (found1 && found2)
        {
          idl_raise(IDL_ERROR,
                    "A SERIOUS ERROR HAS OCCURED !!!!!! Regroup. Try Again.");
        }
    }
}
```

```
/* select_op is executed when Selection is selected within the browser. The
   sytax for the operation allows the user to input the single relation to be
   operated on, attributed name to select upon, the comparison operator, and
   the name of the object that contains the attribute value for comparison.
   The comparison object must be an instantiation of an existing relation in
   the database. It must have only one tuple that has one attribute filled
   with values, the attribute that you want to use for comparison. Then, the
   relation being operated on is found and each tuple has the specified attr
   compared with the comparison objects attr value using that relations
   comparison operator that was specified in the query.

   Select_op calls the functions: Select_parse_action, report_select_error,
   and init_temp_rel. */
*****************************************************************************
```

Select Method

```
*****************************************************************************

idl_routine void select_op(relation)
     relational_relation relation;
{
  relational_relation ptr_R1,ptr_Attr,ptr_Obj,temp_relation;
  relational_database database;
  idl_trans_mode tmode;
  idl_univ root;
  integer operator;
  string parameter1,attr,comp_obj,comp_opr;  /* references to the parameters
                                                R1, attr,comparison object,
                                                and comparison operator
                                                respectively */
  idl_transaction tr;
  boolean found1,found2,attr_found;
  boolean is_writable = false,select = false;
  char *attr_ptr,*delimeter = ",";  /* delimeter between elements
                                       in attribute list - no spaces allowed */

  integer i=1,count,size,index=0,ii=0;
  idl_linked_elem(relational_tuple) comp_tuple;

  tr = idl_get_trans(relation);
  tmode = idl_trans_mode_default;
  root = idl_trans_get_root(tr);
  database = idl_to(relational_database,root);
  found1 = false;
  found2 = false;
  attr_found = false;
  is_writable = (idl_trans_write_count(tr) > 0);

  brw_input("Select Query",
            "Please input the Select query (R1 select attr comp_op object): ",
            0L,0L,0L,false,
            Select_parse_action);

  /* copy the C strings R1, Obj and Attr into IDL strings */
  parameter1 = idl_copy_string(tr,R1);
  comp_obj = idl_copy_string(tr,Obj);
  attr = idl_copy_string(tr,Attr);

  if (strcmp(Comp_opr,"=") == 0)
    operator = 1;
  else if (strcmp(Comp_opr,">") == 0)
    operator = 2;
  else if (strcmp(Comp_opr,"<") == 0)
    operator = 3;
  else if (strcmp(Comp_opr,"/=") == 0)
    operator = 4;
```

```
else if (strcmp(Comp_opr,">=") == 0)
  operator = 5;
else if (strcmp(Comp_opr,"<=") == 0)
  operator = 6;
else
  idl_raise(IDL_ERROR,
            "It is hard to believe that this error could occure!");

/* search the database for the two relations: R1 and Obj */
idl_linked_for (relational_relation,database->relations,rel)
  {
    if (strcmp (rel->relation_name,parameter1) == 0)
      /* found relation 1 */
      {
        ptr_R1 = rel; /* point at relation 1 */
        found1 = true;
      }

    if (strcmp (rel->relation_name,comp_obj) == 0)
      /* found the comparison object */
      {
        ptr_Obj = rel;
        found2 = true;
      }
  } idl_end_for


/* check the attr name in the of the select operation to
   ensure that the field exists in the relation R1 */

idl_array_for(relational_name,ptr_R1->attribute_names,aname)
  {
    ii++; /* keep track of position of attribute in the relation schema -
             that is, the attribute list */
    if (strcmp (aname->name,attr) == 0) /* attr name in attr list is a field
                                           of R1 */
      {
        attr_found = true;
        index = ii;
        break;
      }
  } idl_end_for

if (found1 && found2 && attr_found)
  {
    /* Everything is ok, perform select operation on relation R1.
       Iterate through every tuple of R1 and compare the field named in
       the 'attr' variable with the values of the object specified in
       comp_obj. using the specified comparison operator in Comp_opr. */

    temp_relation = init_temp_rel(ptr_R1,ptr_Obj);

    idl_linked_for(relational_tuple,ptr_Obj->tuples,ctuple)
      {
        idl_linked_for(relational_tuple,ptr_R1->tuples,r1_tuple)
          {
            select = false;
            switch (operator)
              {
              case 1: /* = */
                select = idl_vop(r1_tuple,relational_tuple,equal_to,
                                 (r1_tuple,ctuple,ii));
                break;
              case 2: /* > */
                select = idl_vop(r1_tuple,relational_tuple,greater_than,
                                 (r1_tuple,ctuple,ii));
                break;
              case 3: /* < */
```

```
                    select = idl_vop(r1_tuple,relational_tuple,less_than,
                                     (r1_tuple,ctuple,ii));
                    break;
                  default:
                    idl_raise(IDL_ERROR,
                              "Problems with operator in select operation");
                    break;
                  }

                if (select)
                    idl_insert_back(relational_tuple,
                                    temp_relation->tuples,r1_tuple);

              } idl_end_for
            } idl_end_for

        idl_insert_back(relational_relation,database->relations,temp_relation);
      }
    else
      {
        report_project_error(found1,found2,attr_found);
      }
}
```

```
/*****************************************************************************
*
*    Class tuple methods
*
*        equal_to
*        less_than
*        greater_than
*
*        initialize_tuple
*        insert_fields
*        insert_tuples
*
*****************************************************************************/
*****************************************************************************

                              Class Tuple Methods
*****************************************************************************

/* this is the default duplicate tuple check function */

idl_routine boolean equal_to(r1_tuple,r2_tuple,index)
     relational_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  /* both of these are pointers, so if they point to the same
     object, then they are identical tuples. However, two tuples
     that don't point to the same objects may have the same values
     for the values of each of their individual objects. */

  if (r1_tuple == r2_tuple)
    return true;
  else
    return false;
}

/*===========================================================================*/

/* this function is called by create_tuple of class relation if there has been
   no redefinition of this method by a subclass of tuple. If this function is
   executed, then there is an error since it must be over-written. */

idl_routine relational_tuple initialize_tuple(tuple_type)
     relational_tuple tuple_type;
{
  relational_tuple new_tuple;
  idl_transaction tr = idl_get_trans(tuple_type);

  idl_raise(IDL_ERROR,
            "There is no method for initializing a tuple of\nthe type that this
relation contains.");

}

/*===========================================================================*/

/* the default insert field inserts all fields of tuple into result_tuple
   by reference */

idl_routine relational_relation insert_fields(relation,result_rel)
     relational_relation relation,result_rel;
{
  idl_transaction tr = idl_get_trans(relation);

  return result_rel = relation;
```

```
}

/*=============================================================================*/

/* the default insert tuple inserts a tuple into result_tuple by reference */

idl_routine relational_relation insert_tuples(relation,result_rel)
     relational_relation relation,result_rel;
{
  idl_transaction tr = idl_get_trans(relation);

  return result_rel = relation;

}

/*=============================================================================*/


idl_routine boolean less_than(r1_tuple,r2_tuple,index)
     relational_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  idl_raise(IDL_ERROR,
          "No less than method has been specified for these relations.\nThe user
must provide them. No default can be provided.");
}


/*=============================================================================*/


idl_routine boolean greater_than(r1_tuple,r2_tuple,index)
     relational_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  idl_raise(IDL_ERROR,
          "No greater than method has been specified for these relations.\nThe
user must provide them. No default can be provided.");
}
```

```
/*****************************************************************************
*
*    Class name methods
*
*        name_key
*        name_print
*
*****************************************************************************/

idl_routine void name_key(name)
     relational_name name;
{
  idl_univ u;
  if (idl_string_size(name->name) == 0)
    {
      dpy_cstring("** Unnamed/No Attributes **");
    }
  else
    {
      u = idl_to(idl_univ,name->name);
      idl_vop(u,idl_univ,idl_key,(u));
    }
}

/*===========================================================================*/

idl_routine void name_print(name,mode)
     relational_name name;
     dpy_dmode mode;
{
  idl_transaction tr = idl_get_trans(name);
  boolean can_write = idl_trans_write_count(tr) > 0;
  dpy_dmode mode1;

  mode1 = mode;
  mode1.embed = 1;

  if (can_write) mode1.expand = 1;

  if (mode1.expand > 1)
    {
      idl_top(idl_any,idl_print,(name,mode));
    }

  dpy_attr(relational_name,name,name,mode1);
  dpy_eol();
}
```

```
/*****************************************************************************
*
*    Class emp_tuple methods
*
*        emp_tuple_key
*        emp_tuple_print
*        emp_equal_to
*        initialize_emp_tuple
*
*****************************************************************************/
*****************************************************************************

                              Class Emp_Tuple Methods
*****************************************************************************

idl_routine void emp_tuple_key(tuple)
     relational_emp_tuple tuple;
{
  idl_univ u;
  if (idl_string_size(tuple->person->lname) == 0)
    {
      dpy_cstring("** Unnamed Tuple **");
    }
  else
    {
      u = idl_to(idl_univ,tuple->person->lname);
      idl_vop(u,idl_univ,idl_key,(u));
    }
}

/*===========================================================================*/

idl_routine void emp_tuple_print(emp_tuple,mode)
     relational_emp_tuple emp_tuple;
     dpy_dmode mode;
{
  idl_transaction tr = idl_get_trans(emp_tuple);
  boolean can_write = idl_trans_write_count(tr) > 0;
  dpy_dmode mode1;

  mode1 = mode;
  mode1.embed = 1;

  if (can_write) mode1.expand = 1;

  if (mode1.expand > 1)
    {
      idl_top(idl_any,idl_print,(emp_tuple,mode));
    }

  dpy_attr(relational_emp_tuple,emp_tuple,person,mode1);
  dpy_eol();
  dpy_spacey(2L);
  dpy_eol();
  dpy_attr(relational_emp_tuple,emp_tuple,phone,mode1);
  dpy_eol();
  dpy_spacey(2L);
  dpy_eol();
  dpy_attr(relational_emp_tuple,emp_tuple,address,mode1);
  dpy_eol();

  /* add widget in later */

}

/*===========================================================================*/
```

```
/* this function returns true if r1 tuple is identical to r2 tuple and
   returns false otherwise. */

idl_routine boolean emp_equal_to(r1_tuple,r2_tuple,index)
     relational_emp_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  /* both of these are pointers, so if they point to the same
     object, then they are identical tuples. However, two tuples
     that don't point to the same objects may have the same values
     for the values of each of their individual objects. */

  if (index == 0) /* compare the entire tuple - all attributes */
    {
      if (r1_tuple == r2_tuple)
        return true;
      else /* not the same objects but need to check attribute values */
        {
          /* need to check each attr value. The first time one is found
             that does not have the same value, the function stops and
             returns not dulicate (false). Only if all attr are identical
             does the function return duplicate (true). */

          if (!strcmp(r1_tuple->person->fname,r2_tuple->person->fname) &&
              !strcmp(r1_tuple->person->mname,r2_tuple->person->mname) &&
              !strcmp(r1_tuple->person->lname,r2_tuple->person->lname) &&
              !strcmp(r1_tuple->person->bdate,r2_tuple->person->bdate) &&
              r1_tuple->person->ssn ==  r2_tuple->person->ssn &&
              !strcmp(r1_tuple->person->spouse,r2_tuple->person->spouse) &&
              !strcmp(r1_tuple->address->street,r2_tuple->address->street) &&
              !strcmp(r1_tuple->address->city,r2_tuple->address->city) &&
              !strcmp(r1_tuple->address->state,r2_tuple->address->state) &&
              !strcmp(r1_tuple->address->zip,r2_tuple->address->zip) &&
              !strcmp(r1_tuple->phone->number,r2_tuple->phone->number))
            return true;
          else
            return false;
        }
    }
  else /* compare only the attribute specified by the index */
    {
      switch (index)
        {
        case 1:
          if (!strcmp(r1_tuple->person->fname,r2_tuple->person->fname) &&
              !strcmp(r1_tuple->person->mname,r2_tuple->person->mname) &&
              !strcmp(r1_tuple->person->lname,r2_tuple->person->lname) &&
              !strcmp(r1_tuple->person->bdate,r2_tuple->person->bdate) &&
              r1_tuple->person->ssn == r2_tuple->person->ssn &&
              !strcmp(r1_tuple->person->spouse,r2_tuple->person->spouse))
            return true;
          else
            return false;
          break;
        case 2:
          if (!strcmp(r1_tuple->address->street,r2_tuple->address->street) &&
              !strcmp(r1_tuple->address->city,r2_tuple->address->city) &&
              !strcmp(r1_tuple->address->state,r2_tuple->address->state) &&
              !strcmp(r1_tuple->address->zip,r2_tuple->address->zip))
            return true;
          else
            return false;
          break;
        case 3:
          if (!strcmp(r1_tuple->phone->number,r2_tuple->phone->number))
            return true;
```

```
          else
            return false;
          break;
        case 4:
          idl_raise(IDL_ERROR,
                    "Widget is not defined and thus can't be compared!");
          break;
        default:
          idl_raise(IDL_ERROR,
                    "An ugly error has occured in emp_equal_to");
          break;
        }
    }
}

/*=============================================================================*/

/* this function returns true if r1 tuple is less than r2 tuple and
   returns false otherwise. */

idl_routine boolean emp_less_than(r1_tuple,r2_tuple,index)
     relational_emp_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  switch (index)
    {
    case 1:
      if ((strcmp(r1_tuple->person->fname,r2_tuple->person->fname) < 0) &&
          (strcmp(r1_tuple->person->mname,r2_tuple->person->mname) < 0) &&
          (strcmp(r1_tuple->person->lname,r2_tuple->person->lname) < 0) &&
          (strcmp(r1_tuple->person->bdate,r2_tuple->person->bdate) < 0) &&
          r1_tuple->person->ssn < r2_tuple->person->ssn &&
          (strcmp(r1_tuple->person->spouse,r2_tuple->person->spouse) < 0))
        return true;
      else
        return false;
      break;
    case 2:
      if ((strcmp(r1_tuple->address->street,r2_tuple->address->street) < 0) &&
          (strcmp(r1_tuple->address->city,r2_tuple->address->city) < 0) &&
          (strcmp(r1_tuple->address->state,r2_tuple->address->state) < 0) &&
          (strcmp(r1_tuple->address->zip,r2_tuple->address->zip) < 0))
        return true;
      else
        return false;
      break;
    case 3:
      if ((strcmp(r1_tuple->phone->number,r2_tuple->phone->number) < 0))
        return true;
      else
        return false;
      break;
    case 4:
      idl_raise(IDL_ERROR,
                "Widget is not defined and thus can't be compared!");
      break;
    default:
      idl_raise(IDL_ERROR,
                "An ugly error has occured in emp_less_than");
      break;
    }
}

/*=============================================================================*/

/* this function returns true if r1 tuple is greater than r2 tuple and
```

```
   returns false otherwise. */

idl_routine boolean emp_greater_than(r1_tuple,r2_tuple,index)
     relational_emp_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  switch (index)
    {
    case 1:
      if ((strcmp(r1_tuple->person->fname,r2_tuple->person->fname) > 0) &&
          (strcmp(r1_tuple->person->mname,r2_tuple->person->mname) > 0) &&
          (strcmp(r1_tuple->person->lname,r2_tuple->person->lname) > 0) &&
          (strcmp(r1_tuple->person->bdate,r2_tuple->person->bdate) > 0) &&
          r1_tuple->person->ssn > r2_tuple->person->ssn &&
          (strcmp(r1_tuple->person->spouse,r2_tuple->person->spouse) > 0))
        return true;
      else
        return false;
      break;
    case 2:
      if ((strcmp(r1_tuple->address->street,r2_tuple->address->street) > 0) &&
          (strcmp(r1_tuple->address->city,r2_tuple->address->city) > 0) &&
          (strcmp(r1_tuple->address->state,r2_tuple->address->state) > 0) &&
          (strcmp(r1_tuple->address->zip,r2_tuple->address->zip) > 0))
        return true;
      else
        return false;
      break;
    case 3:
      if ((strcmp(r1_tuple->phone->number,r2_tuple->phone->number) > 0))
        return true;
      else
        return false;
      break;
    case 4:
      idl_raise(IDL_ERROR,
                "Widget is not defined and thus can't be compared!");
      break;
    default:
      idl_raise(IDL_ERROR,
                "An ugly error has occured in emp_greater_than");
      break;
    }
}

/*============================================================================*/

/* this function is called from create_tuple. It redefines the implementation
   for the class tuples method initialize_tuples. Specifically, a new employee
   tuple is created and given initial values (all of which are legal).
   It returns a tuple to create_tuple which then allows the user to initialize
   this tuple with values. Each subtype of tuple needs to have a function
   like this. */

idl_routine relational_tuple initialize_emp_tuple(tuple_type)
     relational_tuple tuple_type;
{
  relational_emp_tuple new_tuple;
  idl_transaction tr = idl_get_trans(tuple_type);
  string empty = idl_copy_string(tr,"");

  new_tuple = idl_new(tr,relational_emp_tuple); /* must still assign
                                                   legal values*/

  new_tuple->person = idl_new(tr,relational_person);
  new_tuple->person->fname = empty;
```

```
    new_tuple->person->mname = empty;
    new_tuple->person->lname = empty;
    new_tuple->person->bdate = empty;
    new_tuple->person->ssn = 0;
    new_tuple->person->spouse = empty;

    new_tuple->address = idl_new(tr,relational_addr);
    new_tuple->address->street = empty;
    new_tuple->address->city = empty;
    new_tuple->address->state = empty;
    new_tuple->address->zip = empty;

    new_tuple->phone = idl_new(tr,relational_phone_number);
    new_tuple->phone->number = empty;

    return idl_to(relational_tuple,new_tuple);

}
```

```
/*****************************************************************************
*
*    Class person methods
*
*        person_key
*        person_print
*
*****************************************************************************/

idl_routine void person_key(person)
     relational_person person;
{
  boolean b1,b2,b3;
  b1 = rexists(person,fname);
  b2 = rexists(person,mname);
  b3 = rexists(person,lname);
  if (b1)
    {
      dpy_cstring(person->fname);
      if (b2 || b3)
        {
          dpy_spacex(1L);
        }
    }
  if (b2)
    {
      dpy_cstring(person->mname);
      if (b3)
        {
          dpy_spacex(1L);
        }
    }
  if (b3)
    {
      dpy_cstring(person->lname);
    }
  if (! b1 && ! b2 && ! b3)
    {
      dpy_cstring("** No name **");
    }
}

/*===========================================================================*/

idl_routine void person_print(person,mode)
     relational_person person;
     dpy_dmode mode;
{
  boolean b1,b2,b3;

  if (mode.expand > 1)
    {
      idl_top(idl_any,idl_print,(person,mode));
      return;
    }
  b1 = dexists(person,fname);
  b2 = dexists(person,mname);
  b3 = dexists(person,lname);
  if (mode.expand > 0) dpy_spacey(1L);
  if (b1)
    {
      dpy_attr(relational_person,person,fname,mode);
      if (b2 || b3)
        {
          dpy_spacex(1L);
        }
    }
  if (b2)
```

```
    {
      dpy_attr(relational_person,person,mname,mode);
      if (b3)
        {
          dpy_spacex(1L);
        }
    }
  if (b3)
    {
      dpy_attr(relational_person,person,lname,mode);
    }
  if (! b1 && ! b2 && ! b3)
    {
      dpy_cstring("** No name **");
    }
  if (dexists(person,bdate))
    {
      dpy_spacex(1L);
      dpy_cstring("(");
      dpy_attr(relational_person,person,bdate,mode);
      dpy_cstring(")");
    }
  else
    {
      dpy_cstring("  ** No Birth Date Entered **");
    }
  if (dexists(person,spouse) || pdexists(person,sptr))
    {
      if (mode.expand > 0 || ! pdexists(person,sptr))
        {
          dpy_cstring(" [");
          dpy_attr(relational_person,person,spouse,mode);
          dpy_cstring("]");
        }
      if (pdexists(person,sptr))
        {
          if (idl_get_display_embed(relational_person,sptr))
            {
              dpy_eol();
              dpy_attr(relational_person,person,sptr,mode);
            }
          else
            {
              dpy_cstring(" [");
              dpy_attr(relational_person,person,sptr,mode);
              dpy_cstring("]");
            }
        }
    }
  dpy_eol();
  dpy_spacey(1L);
  dpy_cstring("SSN: ");
  dpy_attr(relational_person,person,ssn,mode);
  dpy_eol();
}
```

```
/*****************************************************************************
*
*    Class addr methods
*
*        addr_key
*        addr_print
*
*****************************************************************************/


idl_routine void addr_key(address)
     relational_addr address;
(
  if (idl_string_size(address->state) == 0)
    (
      dpy_cstring("** No Street Address **");
      dpy_eol();
    }
  else
    (
      dpy_cstring(address->street);
      dpy_eol();
    }
}

/*===========================================================================*/

idl_routine void addr_print(address,mode)
     relational_addr address;
     dpy_dmode mode;
(
  idl_transaction tr = idl_get_trans(address);
  boolean can_write = idl_trans_write_count(tr) > 0;

  if (can_write) mode.expand = 1;

  dpy_attr(relational_addr,address,street,mode);
  dpy_eol();
  dpy_spacey(1L);
  dpy_attr(relational_addr,address,city,mode);

  if ((mode.expand > 0 ||
       (idl_get_display(relational_addr,state) &&
        (address->state != 0 &&
         (idl_string_size(address->state)) != 0))))
    dpy_cstring(",");

  dpy_spacex(1L);
  dpy_attr(relational_addr,address,state,mode);
  dpy_eol();
  dpy_spacey(1L);
  dpy_spacex(15L);
  dpy_attr(relational_addr,address,zip,mode);
  dpy_eol();
)
```

```
/*****************************************************************************
*
*    Class phone_number methods
*
*        phone_number_print
*
*****************************************************************************/


idl_routine void phone_number_print(pnumber,mode)
     relational_phone_number pnumber;
     dpy_dmode mode;
{
  dpy_attr(relational_phone_number,pnumber,number,mode);
  dpy_eol();
}
```

```
/*****************************************************************************
*
*    Class proj_tuple methods
*
*        proj_tuple_key
*        proj_tuple_print
*        initialize_proj_tuple
*        proj_equal_to
*        proj_less_than
*        proj_greater_than
*
*****************************************************************************/
*******************************************************************************

                            Class Proj_tuple Methods
*******************************************************************************


idl_routine void proj_tuple_key(proj_tuple)
     relational_proj_tuple proj_tuple;
{
  dpy_integer(proj_tuple->essn,9);
  dpy_spacex(4L);
  dpy_integer(proj_tuple->proj_num,3);
  dpy_spacex(4L);
  dpy_rational(proj_tuple->hours,6);
  dpy_eol();
}

/*============================================================================*/

idl_routine void proj_tuple_print(proj_tuple,mode)
     relational_proj_tuple proj_tuple;
     dpy_dmode mode;
{
  idl_transaction tr = idl_get_trans(proj_tuple);
  boolean can_write = idl_trans_write_count(tr) > 0;

  if (can_write) mode.expand = 1;

  dpy_attr(relational_proj_tuple,proj_tuple,essn,mode);
  dpy_eol();
  dpy_spacey(1L);
  dpy_attr(relational_proj_tuple,proj_tuple,proj_num,mode);
  dpy_eol();
  dpy_spacey(1L);
  dpy_attr(relational_proj_tuple,proj_tuple,hours,mode);
  dpy_eol();
}

/*============================================================================*/

/* this function is called from create_tuple. It redefines the implementation
   for the class tuples method initialize_tuples. Specifically, a new project
   tuple is created and given initial values (all of which are legal).
   It returns a tuple to create_tuple which then allows the user to initialize
   this tuple with values. Each subtype of tuple needs to have a function
   like this. */

idl_routine relational_tuple initialize_proj_tuple(tuple_type)
     relational_tuple tuple_type;
{
  relational_proj_tuple new_tuple;
  idl_transaction tr = idl_get_trans(tuple_type);
  string empty = idl_copy_string(tr,"");

  new_tuple = idl_new(tr,relational_proj_tuple);
  /* must still assign legal values*/
```

```
  new_tuple->essn = 0;
  new_tuple->proj_num = 0;
  new_tuple->hours = 0.0;

  return idl_to(relational_tuple,new_tuple);
}

/*=============================================================================*/

/* this function returns true if r1 tuple is equal to r2 tuple and
   returns false otherwise. */

idl_routine boolean proj_equal_to(r1_tuple,r2_tuple,index)
     relational_proj_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  switch (index)
    {
    case 1:
      if (r1_tuple->essn == r2_tuple->essn)
        return true;
      else
        return false;
      break;
    case 2:
      if (r1_tuple->proj_num == r2_tuple->proj_num)
        return true;
      else
        return false;
      break;
    case 3:
      if (r1_tuple->hours == r2_tuple->hours)
        return true;
      else
        return false;
      break;
    default:
      idl_raise(IDL_ERROR,
                "An ugly error has occured in proj_equal_to");
      break;
    }
}

/*=============================================================================*/

/* this function returns true if r1 tuple is less than r2 tuple and
   returns false otherwise. */

idl_routine boolean proj_less_than(r1_tuple,r2_tuple,index)
     relational_proj_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  switch (index)
    {
    case 1:
      if (r1_tuple->essn < r2_tuple->essn)
        return true;
      else
        return false;
      break;
    case 2:
      if (r1_tuple->proj_num < r2_tuple->proj_num)
        return true;
```

```
      else
        return false;
      break;
    case 3:
      if (r1_tuple->hours < r2_tuple->hours)
        return true;
      else
        return false;
      break;
    default:
      idl_raise(IDL_ERROR,
                "An ugly error has occured in proj_less_than");
      break;
    )
)

/*=============================================================================*/

/* this function returns true if r1 tuple is greater than r2 tuple and
   returns false otherwise. */

idl_routine boolean proj_greater_than(r1_tuple,r2_tuple,index)
     relational_proj_tuple r1_tuple,r2_tuple;
     integer index;
{
  idl_transaction tr = idl_get_trans(r1_tuple);

  switch (index)
    {
    case 1:
      if (r1_tuple->essn > r2_tuple->essn)
        return true;
      else
        return false;
      break;
    case 2:
      if (r1_tuple->proj_num > r2_tuple->proj_num)
        return true;
      else
        return false;
      break;
    case 3:
      if (r1_tuple->hours > r2_tuple->hours)
        return true;
      else
        return false;
      break;
    default:
      idl_raise(IDL_ERROR,
                "An ugly error has occured in proj_greater_than");
      break;
    )
)
```

```
/*****************************************************************************
*
*    Class cart1_result_tuple methods
*
*        cart1_result_tuple_key
*        cart1_result_tuple_print
*        initialize_cart1_resul_tuple
*        insert_cart1_result_tuples
*
*****************************************************************************/
*****************************************************************************

                        Class Cart1_Result_Tuple Methods
*****************************************************************************


idl_routine void cart1_result_tuple_key(cart1_tuple)
     relational_cart1_result_tuple cart1_tuple;
{
  idl_univ u;

  u = idl_to(idl_univ,cart1_tuple->person->lname);
  idl_vop(u,idl_univ,idl_key,(u));
  dpy_spacex(4L);
  dpy_integer(cart1_tuple->essn,9);
  dpy_spacex(4L);
  dpy_integer(cart1_tuple->proj_num,3);
  dpy_spacex(4L);
  dpy_rational(cart1_tuple->hours,6);
  dpy_eol();
}

/*===========================================================================*/

idl_routine void cart1_result_tuple_print(cart1_tuple,mode)
     relational_cart1_result_tuple cart1_tuple;
     dpy_dmode mode;
{
  idl_transaction tr = idl_get_trans(cart1_tuple);
  boolean can_write = idl_trans_write_count(tr) > 0;

  if (can_write) mode.expand = 1;

  mode.embed = 1;

  if (mode.expand > 1)
    {
      idl_top(idl_any,idl_print,(cart1_tuple,mode));
    }

  dpy_attr(relational_emp_tuple,cart1_tuple,person,mode);
  dpy_eol();
  dpy_spacey(1L);
  dpy_eol();
  dpy_attr(relational_emp_tuple,cart1_tuple,phone,mode);
  dpy_eol();
  dpy_spacey(1L);
  dpy_eol();
  dpy_attr(relational_emp_tuple,cart1_tuple,address,mode);
  dpy_eol();
  dpy_spacey(1L);

  /* add widget in later when it is defined as something */

  dpy_attr(relational_cart1_result_tuple,cart1_tuple,essn,mode);
  dpy_eol();
  dpy_spacey(1L);
  dpy_attr(relational_cart1_result_tuple,cart1_tuple,proj_num,mode);
```

```
  dpy_eol();
  dpy_spacey(1L);
  dpy_attr(relational_cart1_result_tuple,cart1_tuple,hours,mode);
  dpy_eol();
}

/*=============================================================================*/

/* this function is called from create_tuple. It redefines the implementation
   for the class tuples method initialize_tuples. Specifically, a new project
   tuple is created and given initial values (all of which are legal).
   It returns a tuple to create_tuple which then allows the user to initialize
   this tuple with values. Each subtype of tuple needs to have a function
   like this. */

idl_routine relational_tuple initialize_cart1_resul_tuple(tuple_type)
     relational_tuple tuple_type;
{
  relational_cart1_result_tuple new_tuple;
  idl_transaction tr = idl_get_trans(tuple_type);
  string empty = idl_copy_string(tr,"");

  new_tuple = idl_new(tr,relational_cart1_result_tuple);
  /* must still assign legal values*/

  new_tuple->person = idl_new(tr,relational_person);
  new_tuple->person->fname = empty;
  new_tuple->person->mname = empty;
  new_tuple->person->lname = empty;
  new_tuple->person->bdate = empty;
  new_tuple->person->ssn = 0;
  new_tuple->person->spouse = empty;

  new_tuple->address = idl_new(tr,relational_addr);
  new_tuple->address->street = empty;
  new_tuple->address->city = empty;
  new_tuple->address->state = empty;
  new_tuple->address->zip = empty;

  new_tuple->phone = idl_new(tr,relational_phone_number);
  new_tuple->phone->number = empty;

  /* new_tuple->widget needs to have widget defined first */

  new_tuple->essn = 0;
  new_tuple->proj_num = 0;
  new_tuple->hours = 0.0;

  return idl_to(relational_tuple,new_tuple);
}

/*=============================================================================*/

static relational_relation insert_cart1_result_tuples(r1,r2,r3)
     relational_relation r1,r2,r3;
{
  relational_cart1_result_tuple new_tuple;
  relational_emp_tuple rel1;
  relational_proj_tuple rel2;
  idl_transaction tr = idl_get_trans(r1);

  /* get rid of any tuples that may be in the resultant relation structure
     prior to insert the new result */

  r3->tuples = idl_empty_linked(tr,relational_tuple);

  idl_linked_for (relational_tuple,r1->tuples,r1_tuple)
    {
```

```
      idl_linked_for (relational_tuple,r2->tuples,r2_tuple)
        {
          /* send message to get a new tuple created with valid default
             values. */

          new_tuple = idl_to(relational_cart1_result_tuple,
                      idl_vop(r3->tuple_type,relational_tuple, initialize_tuple,
(r3->tuple_type)));

          rel1 = idl_to(relational_emp_tuple,r1_tuple);
          rel2 = idl_to(relational_proj_tuple,r2_tuple);

          new_tuple->person->fname = rel1->person->fname;
          new_tuple->person->mname = rel1->person->mname;
          new_tuple->person->lname = rel1->person->lname;
          new_tuple->person->bdate = rel1->person->bdate;
          new_tuple->person->ssn = rel1->person->ssn;

          new_tuple->address->street = rel1->address->street;
          new_tuple->address->city = rel1->address->city;
          new_tuple->address->state = rel1->address->state;
          new_tuple->address->zip = rel1->address->zip;

          new_tuple->phone->number = rel1->phone->number;

          /* new_tuple->widget needs to have widget defined first */

          new_tuple->essn = rel2->essn;
          new_tuple->proj_num = rel2->proj_num;
          new_tuple->hours = rel2->hours;

          idl_insert_back(relational_tuple,r3->tuples,new_tuple);

        } idl_end_for
    } idl_end_for

  return r3;

}
```

```
/*****************************************************************************
*
*    Class project1_result_tuple methods
*
*        project1_result_tuple_key
*        project1_result_tuple_print
*        initialize_project1_resul_tuple
*        insert_project1_result_flds
*
*****************************************************************************/
*****************************************************************************

                    Class Project1_Result_Tuple Methods
*****************************************************************************


idl_routine void project1_result_tuple_key(project1_result_tuple)
     relational_project1_result_tuple project1_result_tuple;
{
  dpy_rational(project1_result_tuple->hours,6);
  dpy_spacex(4L);
  dpy_integer(project1_result_tuple->essn,9);
  dpy_eol();
}

/*===========================================================================*/

idl_routine void project1_result_tuple_print(project1_result_tuple,mode)
     relational_project1_result_tuple project1_result_tuple;
     dpy_dmode mode;
{
  idl_transaction tr = idl_get_trans(project1_result_tuple);
  boolean can_write = idl_trans_write_count(tr) > 0;

  if (can_write) mode.expand = 1;

  dpy_attr(relational_project1_result_tuple,project1_result_tuple,hours,mode);
  dpy_eol();
  dpy_spacey(1L);
  dpy_attr(relational_project1_result_tuple,project1_result_tuple,essn,mode);
  dpy_eol();
}

/*===========================================================================*/

idl_routine relational_tuple initialize_project1_resul_tuple(tuple_type)
     relational_tuple tuple_type;
{
  relational_project1_result_tuple new_tuple;
  idl_transaction tr = idl_get_trans(tuple_type);

  new_tuple = idl_new(tr,relational_project1_result_tuple);
  /* must still assign legal values*/

  new_tuple->essn = 0;
  new_tuple->hours = 0.0;

  return idl_to(relational_tuple,new_tuple);
}


/*===========================================================================*/

idl_routine relational_relation insert_project1_result_flds(rel,result_rel)
     relational_relation rel,result_rel;
{
  relational_project1_result_tuple new_tuple;
  relational_proj_tuple rel1;
```

```
  idl_transaction tr = idl_get_trans(rel);

  result_rel->tuples = idl_empty_linked(tr,relational_tuple);

  new_tuple = idl_new(tr,relational_project1_result_tuple);
  /* must still assign legal values*/


  idl_linked_for (relational_tuple,rel->tuples,r1_tuple)
    {
      /* send message to get a new tuple created with valid default
         values. */

      new_tuple = idl_to(relational_project1_result_tuple,
                         idl_vop(result_rel-
>tuple_type,relational_tuple,initialize_tuple,(result_rel->tuple_type)));

      rel1 = idl_to(relational_proj_tuple,r1_tuple);

      new_tuple->essn = rel1->essn;
      new_tuple->hours = rel1->hours;

      idl_insert_back(relational_tuple,result_rel->tuples,new_tuple);

    } idl_end_for

  return result_rel;

)
```

```
/*****************************************************************************
*
*    General functions to display a character string and an integer
*
*        exit_action
*        char_screen
*        integer_screen
*
*****************************************************************************/

/* called by char_screen and integer_screen to exit the pop up window */

idl_routine void exit_action(i)
   integer i;
{
   dpy_quit();
}

/*===========================================================================*/

/* display the string s on a pop-up screen using dpy_active(char_screen,s)
   Note: the browser restricts s to 80 characters */

idl_routine void char_screen(s,x,y)
   string s;
   integer x,y;
{
   dpy_open("Display String",true);
   dpy_open("format",true);
   brw_cmd("Exit","exit this screen",exit_action,0,BRW_SCREEN);
   dpy_eol();
   dpy_spacey(2);
   dpy_eol();
   dpy_cstring(s);
   dpy_spacey(2);
   dpy_eol();
   dpy_close();
   brw_input_area(y-3,true);
   dpy_close();
   dpy_boxed(x,y);
}

/*===========================================================================*/

/* display the integer s on a pop-up screen */
idl_routine void integer_screen(s,x,y)
   integer s;
   integer x,y;
{

   dpy_open("Display Integer",true);
   dpy_open("format",true);
   brw_cmd("Exit","exit this screen",exit_action,0,BRW_SCREEN);
   dpy_eol();
   dpy_spacey(1);
   dpy_eol();
   dpy_integer(s,5);
   dpy_spacey(2);
   dpy_eol();
   dpy_close();
   brw_input_area(y-3,true);
   dpy_close();
   dpy_boxed(x,y);
}
```

```
/*****************************************************************************
*
*    Binding and Initialization
*
*****************************************************************************/


idl_define_ops relational_opbind()
{
  idl_bind_root(relational);

  idl_bind("database_key",database_key);
  idl_bind("database_print",database_print);
  idl_bind("create_relation",create_relation);

  idl_bind("relation_key",relation_key);
  idl_bind("relation_print",relation_print);
  idl_bind("create_tuple",create_tuple);
  idl_bind("ck_union_compatability",ck_union_compatability);

  idl_bind("union_op",union_op);
  idl_bind("cart_prod_op",cart_prod_op);
  idl_bind("set_diff_op",set_diff_op);
  idl_bind("project_op",project_op);
  idl_bind("select_op",select_op);

  idl_bind("equal_to",equal_to);
  idl_bind("less_than",less_than);
  idl_bind("greater_than",greater_than);

  idl_bind("initialize_tuple",initialize_tuple);
  idl_bind("insert_fields",insert_fields);
  idl_bind("insert_tuples",insert_tuples);

  idl_bind("name_key",name_key);
  idl_bind("name_print",name_print);

  idl_bind("emp_tuple_key",emp_tuple_key);
  idl_bind("emp_tuple_print",emp_tuple_print);
  idl_bind("emp_equal_to",emp_equal_to);
  idl_bind("emp_less_than",emp_less_than);
  idl_bind("emp_greater_than",emp_greater_than);
  idl_bind("initialize_emp_tuple",initialize_emp_tuple);

  idl_bind("person_key",person_key);
  idl_bind("person_print",person_print);

  idl_bind("addr_key",addr_key);
  idl_bind("addr_print",addr_print);

  idl_bind("phone_number_print",phone_number_print);

  idl_bind("proj_tuple_key",proj_tuple_key);
  idl_bind("proj_tuple_print",proj_tuple_print);
  idl_bind("proj_equal_to",proj_equal_to);
  idl_bind("proj_less_than",proj_less_than);
  idl_bind("proj_greater_than",proj_greater_than);
  idl_bind("initialize_proj_tuple",initialize_proj_tuple);


  idl_bind("cart1_result_tuple_key",cart1_result_tuple_key);
  idl_bind("cart1_result_tuple_print",cart1_result_tuple_print);
  idl_bind("initialize_cart1_resul_tuple",initialize_cart1_resul_tuple);
  idl_bind("insert_cart1_result_tuples",insert_cart1_result_tuples);


  idl_bind("project1_result_tuple_key",project1_result_tuple_key);
  idl_bind("project1_result_tuple_print",project1_result_tuple_print);
```

```
    idl_bind("initialize_project1_resul_tuple",initialize_project1_resul_tuple);
    idl_bind("insert_project1_result_flds",insert_project1_result_flds);

}
```

APPENDIX D: A SAMPLE R/OODBMS DATABASE ASCII CLUSTER FILE

```
-- Generated by IDB System Version 1.1
@[ cid < 16#e6842701 16#00000000 16#00000036 > ;
    lid_seed 3566 ]@
x355^
x1: 1063@ name[ name 1064@ "person" ]
x2: 1065@ name[ name 1066@ "address" ]
x3: 1067@ name[ name 1068@ "phone" ]
x4: 1069@ name[ name 1070@ "widget" ]
x5: 1257@ name[ name 1267@ "person" ]
x6: 1259@ name[ name 1266@ "addr" ]
x7: 1261@ name[ name 1268@ "phone_number" ]
x8: 1272@ name[ name 1275@ "idl_univ" ]
x9: 1072@ person[ fname x176^ ; mname x177^ ; lname x178^ ; bdate x179^ ; ssn
122121212 ; spouse 3015@ "Lisa" ; sptr nil ]
x10: 1074@ addr[ street x181^ ; city x182^ ; state x183^ ; zip x184^ ]
x11: 1075@ phone_number[ number x186^ ]
x12: 1076@ emp_tuple[ person x9^ ; address x10^ ; phone x11^ ; widget nil ]
x13: 1078@ person[ fname x201^ ; mname x202^ ; lname x203^ ; bdate x204^ ; ssn
110121212 ; spouse 3024@ "Stephanie" ; sptr nil ]
x14: 1080@ addr[ street x206^ ; city x207^ ; state x208^ ; zip x209^ ]
x15: 1081@ phone_number[ number x211^ ]
x16: 1082@ emp_tuple[ person x13^ ; address x14^ ; phone x15^ ; widget nil ]
x17: 1084@ person[ fname x226^ ; mname x227^ ; lname x228^ ; bdate x229^ ; ssn
220121212 ; spouse 3033@ "Joan" ; sptr nil ]
x18: 1086@ addr[ street x231^ ; city x232^ ; state x233^ ; zip x234^ ]
x19: 1087@ phone_number[ number x236^ ]
x20: 1088@ emp_tuple[ person x17^ ; address x18^ ; phone x19^ ; widget nil ]
x21: 1090@ person[ fname x251^ ; mname x252^ ; lname x253^ ; bdate x254^ ; ssn
120926190 ; spouse 3040@ "Karin" ; sptr nil ]
x22: 1094@ addr[ street x256^ ; city x257^ ; state x258^ ; zip x259^ ]
x23: 1095@ phone_number[ number x261^ ]
x24: 1096@ emp_tuple[ person x21^ ; address x22^ ; phone x23^ ; widget nil ]
x25: 1098@ person[ fname x276^ ; mname x277^ ; lname x278^ ; bdate x279^ ; ssn
123456789 ; spouse 1097@ "" ; sptr nil ]
x26: 1102@ addr[ street x281^ ; city x282^ ; state x283^ ; zip x284^ ]
x27: 1105@ phone_number[ number x286^ ]
x28: 1106@ emp_tuple[ person x25^ ; address x26^ ; phone x27^ ; widget nil ]
x29: 1348@ person[ fname x301^ ; mname x302^ ; lname x303^ ; bdate x304^ ; ssn
991221234 ; spouse 3053@ "Lesa" ; sptr nil ]
x30: 1349@ addr[ street x306^ ; city x307^ ; state x308^ ; zip x309^ ]
x31: 1350@ phone_number[ number x311^ ]
x32: 1347@ emp_tuple[ person x29^ ; address x30^ ; phone x31^ ; widget nil ]
x33: 1354@ person[ fname 1355@ "" ; mname 1356@ "" ; lname 1382@ "" ; bdate 1358@
"" ; ssn 0 ; spouse 1359@ "" ; sptr nil ]
x34: 1360@ addr[ street 1361@ "" ; city 1362@ "" ; state 1363@ "" ; zip 1364@ "" ]
x35: 1365@ phone_number[ number 1366@ "" ]
x36: 1353@ emp_tuple[ person x33^ ; address x34^ ; phone x35^ ; widget nil ]
x37: 1107@ relation[ relation_name 1108@ "r1" ; attribute_names x38^ ;
attribute_types x39^ ; tuples 1110@ < x12^ x16^ x20^ x24^ x28^ x32^ > ; tuple_type
x36^ ; key x353^ ]
x38: 1113@ < x1^ x2^ x3^ x4^ >
x39: 1254@ < x5^ x6^ x7^ x8^ >
x40: 1239@ person[ fname 1240@ "Nancy" ; mname 3059@ "J." ; lname 1241@ "McClellan"
; bdate 3060@ "14 Feb 57" ; ssn 990124444 ; spouse 1238@ "" ; sptr nil ]
x41: 1242@ addr[ street 3062@ "2331 Long St." ; city 3063@ "Monterey" ; state 3064@
"CA" ; zip 3065@ "93940" ]
x42: 1243@ phone_number[ number 3061@ "(203)999-9991" ]
x43: 1244@ emp_tuple[ person x40^ ; address x41^ ; phone x42^ ; widget nil ]
x44: 1221@ person[ fname 3066@ "Leonard" ; mname 3067@ "H." ; lname 1222@ "Tharpe"
; bdate 3068@ "12 Aug 58" ; ssn 110121212 ; spouse 3069@ "Stephanie" ; sptr nil ]
```

```
x45: 1223@ addr[ street 3071@ "432 Caldwell Drive" ; city 2779@ "Monterey" ; state
3072@ "CA" ; zip 3073@ "93940" ]
x46: 1224@ phone_number[ number 3070@ "(408)452-1234" ]
x47: 1225@ emp_tuple[ person x44^ ; address x45^ ; phone x46^ ; widget nil ]
x48: 1227@ person[ fname 3074@ "David" ; mname 3075@ "M." ; lname 1228@ "Nash" ;
bdate 3076@ "21 Jul 65" ; ssn 23551324 ; spouse 3077@ "Tammy" ; sptr nil ]
x49: 1229@ addr[ street 3079@ "2112 Leidig Circle" ; city 2778@ "Monterey" ; state
3080@ "CA" ; zip 3081@ "93940" ]
x50: 1230@ phone_number[ number 3078@ "(408)123-4567" ]
x51: 1231@ emp_tuple[ person x48^ ; address x49^ ; phone x50^ ; widget nil ]
x52: 1368@ person[ fname 1369@ "" ; mname 1370@ "" ; lname 1371@ "" ; bdate 1372@
"" ; ssn 0 ; spouse 1373@ "" ; sptr nil ]
x53: 1374@ addr[ street 1375@ "" ; city 1376@ "" ; state 1377@ "" ; zip 1378@ "" ]
x54: 1379@ phone_number[ number 1380@ "" ]
x55: 1367@ emp_tuple[ person x52^ ; address x53^ ; phone x54^ ; widget nil ]
x56: 1132@ relation[ relation_name 1133@ "r2" ; attribute_names x38^ ;
attribute_types x39^ ; tuples 1135@ < x43^ x47^ x51^ > ; tuple_type x55^ ; key
1137@ "person -> ssn" ]
x57: 1279@ name[ name 1288@ "person" ]
x58: 1281@ name[ name 1289@ "addr" ]
x59: 1283@ name[ name 1290@ "phone_number" ]
x60: 1285@ name[ name 1291@ "diff_type" ]
x61: 1139@ person[ fname 1140@ "Tim" ; mname 3089@ "J." ; lname 1141@ "Kelly" ;
bdate 3090@ "4 Jul 62" ; ssn 22121212 ; spouse 1138@ "" ; sptr nil ]
x62: 1142@ addr[ street 1143@ "345 Bergin" ; city 1144@ "Monterey" ; state 1145@
"CA" ; zip 3092@ "93940" ]
x63: 1146@ phone_number[ number 3091@ "(408)123-4567" ]
x64: 1148@ emp_tuple[ person x61^ ; address x62^ ; phone x63^ ; widget nil ]
x65: 1150@ person[ fname 1151@ "Ronald" ; mname 1152@ "L." ; lname 1153@ "Spear"
; bdate 3082@ "29 Dec 62" ; ssn 120926190 ; spouse 3083@ "Karin" ; sptr nil ]
x66: 1154@ addr[ street 3085@ "397B Ricketts Road" ; city 3086@ "Monterey" ; state
3087@ "CA" ; zip 3088@ "93940" ]
x67: 1155@ phone_number[ number 3084@ "(408)375-8619" ]
x68: 1156@ emp_tuple[ person x65^ ; address x66^ ; phone x67^ ; widget nil ]
x69: 1384@ person[ fname 1385@ "" ; mname 1386@ "" ; lname 1387@ "" ; bdate 1388@
"" ; ssn 0 ; spouse 1389@ "" ; sptr nil ]
x70: 1390@ addr[ street 1391@ "" ; city 1392@ "" ; state 1393@ "" ; zip 1394@ "" ]
x71: 1395@ phone_number[ number 1396@ "" ]
x72: 1383@ emp_tuple[ person x69^ ; address x70^ ; phone x71^ ; widget nil ]
x73: 1157@ relation[ relation_name 1158@ "r3" ; attribute_names x38^ ;
attribute_types 1276@ < x57^ x58^ x59^ x60^ > ; tuples 1160@ < x64^ x68^ > ;
tuple_type x72^ ; key 1162@ "person -> ssn" ]
x74: 1163@ name[ name 1164@ "person -> fname" ]
x75: 1165@ name[ name 1166@ "person -> mname" ]
x76: 1167@ name[ name 1168@ "person -> lname" ]
x77: 1169@ name[ name 1170@ "person -> bdate" ]
x78: 1171@ name[ name 1172@ "person -> spouse" ]
x79: 1173@ name[ name 1174@ "addr -> street" ]
x80: 1175@ name[ name 1176@ "addr -> city" ]
x81: 1177@ name[ name 1178@ "addr -> state" ]
x82: 1179@ name[ name 1180@ "addr -> zip" ]
x83: 1181@ name[ name 1182@ "phone -> number" ]
x84: 1310@ name[ name 1330@ "string" ]
x85: 1312@ name[ name 1331@ "string" ]
x86: 1314@ name[ name 1332@ "string" ]
x87: 1316@ name[ name 1333@ "string" ]
x88: 1318@ name[ name 1340@ "integer" ]
x89: 1320@ name[ name 1335@ "string" ]
x90: 1322@ name[ name 1336@ "string" ]
x91: 1324@ name[ name 1337@ "string" ]
x92: 1326@ name[ name 1338@ "string" ]
x93: 1328@ name[ name 1339@ "string" ]
x94: 1183@ ""
x95: 1184@ person[ fname x94^ ; mname x94^ ; lname 1185@ "Larson" ; bdate x94^ ;
ssn 0 ; spouse x94^ ; sptr nil ]
x96: 1186@ addr[ street x94^ ; city x94^ ; state x94^ ; zip x94^ ]
x97: 1187@ phone_number[ number x94^ ]
x98: 1188@ emp_tuple[ person x95^ ; address x96^ ; phone x97^ ; widget nil ]
```

```
x99: 1189@ ""
x100: 1190@ person[ fname x99^ ; mname x99^ ; lname 1191@ "Johnson" ; bdate x99^
; ssn 0 ; spouse x99^ ; sptr nil ]
x101: 1192@ addr[ street x99^ ; city x99^ ; state x99^ ; zip x99^ ]
x102: 1193@ phone_number[ number x99^ ]
x103: 1194@ emp_tuple[ person x100^ ; address x101^ ; phone x102^ ; widget nil ]
x104: 1195@ ""
x105: 1196@ person[ fname x104^ ; mname x104^ ; lname 1197@ "Lombardo" ; bdate
x104^ ; ssn 0 ; spouse x104^ ; sptr nil ]
x106: 1198@ addr[ street x104^ ; city x104^ ; state x104^ ; zip x104^ ]
x107: 1199@ phone_number[ number x104^ ]
x108: 1200@ emp_tuple[ person x105^ ; address x106^ ; phone x107^ ; widget nil ]
x109: 1440@ person[ fname 1441@ "" ; mname 1442@ "" ; lname 1443@ "" ; bdate 1444@
"" ; ssn 0 ; spouse 1445@ "" ; sptr nil ]
x110: 1446@ addr[ street 1447@ "" ; city 1448@ "" ; state 1449@ "" ; zip 1450@ "" ]
x111: 1451@ phone_number[ number 1452@ "" ]
x112: 1439@ emp_tuple[ person x109^ ; address x110^ ; phone x111^ ; widget nil ]
x113: 1201@ relation[ relation_name 1202@ "test2" ; attribute_names x114^ ;
attribute_types x115^ ; tuples 1204@ < x98^ x103^ x108^ > ; tuple_type x112^ ; key
1206@ "person -> ssn" ]
x114: 1207@ < x74^ x75^ x76^ x77^ x78^ x79^ x80^ x81^ x82^ x83^ >
x115: 1307@ < x84^ x85^ x86^ x87^ x88^ x89^ x90^ x91^ x92^ x93^ >
x116: 1426@ person[ fname 1427@ "" ; mname 1428@ "" ; lname 1429@ "" ; bdate 1430@
"" ; ssn 0 ; spouse 1431@ "" ; sptr nil ]
x117: 1432@ addr[ street 1433@ "" ; city 1434@ "" ; state 1435@ "" ; zip 1436@ "" ]
x118: 1437@ phone_number[ number 1438@ "" ]
x119: 1425@ emp_tuple[ person x116^ ; address x117^ ; phone x118^ ; widget nil ]
x120: 1208@ relation[ relation_name 1209@ "TEMP1" ; attribute_names x114^ ;
attribute_types x115^ ; tuples 1211@ < x98^ x103^ x108^ x64^ > ; tuple_type x119^
; key 1213@ "person -> ssn" ]
x121: 1295@ name[ name 1303@ "person" ]
x122: 1297@ name[ name 1304@ "addr" ]
x123: 1299@ name[ name 1305@ "phone_number" ]
x124: 1301@ name[ name 1306@ "idl_univ" ]
x125: 1215@ person[ fname 3093@ "James" ; mname 3094@ "S." ; lname 1216@ "Baumann"
; bdate 3095@ "12 Jan 85" ; ssn 550121212 ; spouse 1214@ "" ; sptr nil ]
x126: 1217@ addr[ street 3097@ "41112 Lost Lane" ; city 2780@ "Fayetteville" ;
state 3098@ "NC" ; zip 3099@ "32212" ]
x127: 1218@ phone_number[ number 3096@ "(231)222-3333" ]
x128: 1219@ emp_tuple[ person x125^ ; address x126^ ; phone x127^ ; widget nil ]
x129: 1398@ person[ fname 1399@ "" ; mname 1400@ "" ; lname 1401@ "" ; bdate 1402@
"" ; ssn 0 ; spouse 1403@ "" ; sptr nil ]
x130: 1404@ addr[ street 1405@ "" ; city 1406@ "" ; state 1407@ "" ; zip 1408@ "" ]
x131: 1409@ phone_number[ number 1410@ "" ]
x132: 1397@ emp_tuple[ person x129^ ; address x130^ ; phone x131^ ; widget nil ]
x133: 1232@ relation[ relation_name 1233@ "r4" ; attribute_names x38^ ;
attribute_types 1292@ < x121^ x122^ x123^ x124^ > ; tuples 1235@ < x128^ x47^ x51^
> ; tuple_type x132^ ; key 1237@ "person -> ssn" ]
x134: 1412@ person[ fname 1413@ "" ; mname 1414@ "" ; lname 1415@ "" ; bdate 1416@
"" ; ssn 0 ; spouse 1417@ "" ; sptr nil ]
x135: 1418@ addr[ street 1419@ "" ; city 1420@ "" ; state 1421@ "" ; zip 1422@ "" ]
x136: 1423@ phone_number[ number 1424@ "" ]
x137: 1411@ emp_tuple[ person x134^ ; address x135^ ; phone x136^ ; widget nil ]
x138: 1245@ relation[ relation_name 1246@ "r5" ; attribute_names x38^ ;
attribute_types x39^ ; tuples 1248@ < x43^ > ; tuple_type x137^ ; key 1250@ "person
-> ssn" ]
x139: 1501@ name[ name 1518@ "Employee SSN" ]
x140: 1503@ name[ name 1519@ "Project Number" ]
x141: 1505@ name[ name 1520@ "Hours Worked" ]
x142: 1510@ name[ name 1521@ "integer" ]
x143: 1512@ name[ name 1522@ "integer" ]
x144: 1514@ name[ name 1523@ "rational" ]
x145: 1526@ proj_tuple[ essn 120926190 ; proj_num 2 ; hours 10.0 ]
x146: 1529@ proj_tuple[ essn 999999999 ; proj_num 1 ; hours 45.2 ]
x147: 1532@ proj_tuple[ essn 123456789 ; proj_num 2 ; hours 51.5 ]
x148: 1535@ proj_tuple[ essn 987654321 ; proj_num 30 ; hours 4.0 ]
x149: 1497@ proj_tuple[ essn 0 ; proj_num 0 ; hours 0.0 ]
```

```
x150: 1490@ relation[ relation_name 1495@ "pt1" ; attribute_names 1498@ < x139^
x140^ x141^ > ; attribute_types 1507@ < x142^ x143^ x144^ > ; tuples 1494@ < x145^
x146^ x147^ x148^ > ; tuple_type x149^ ; key 1496@ "essn,proj_num" ]
x151: 1554@ name[ name 1575@ "Employee SSN" ]
x152: 1556@ name[ name 1576@ "Project Number" ]
x153: 1558@ name[ name 1578@ "Hours Worked" ]
x154: 1560@ name[ name 1579@ "integer" ]
x155: 1562@ name[ name 1580@ "integer" ]
x156: 1564@ name[ name 1581@ "rational" ]
x157: 1568@ proj_tuple[ essn 550926190 ; proj_num 23 ; hours 34.4 ]
x158: 1571@ proj_tuple[ essn 666666666 ; proj_num 2 ; hours 15.0 ]
x159: 1574@ proj_tuple[ essn 123456789 ; proj_num 1 ; hours 24.0 ]
x160: 1547@ proj_tuple[ essn 0 ; proj_num 0 ; hours 0.0 ]
x161: 1540@ relation[ relation_name 1545@ "pt2" ; attribute_names 1548@ < x151^
x152^ x153^ > ; attribute_types 1551@ < x154^ x155^ x156^ > ; tuples 1544@ < x157^
x158^ x159^ > ; tuple_type x160^ ; key 1546@ "essn,proj_num" ]
x162: 1618@ name[ name 1658@ "Person" ]
x163: 1620@ name[ name 1659@ "Address" ]
x164: 1622@ name[ name 1660@ "Phone" ]
x165: 1624@ name[ name 1661@ "Widget" ]
x166: 1626@ name[ name 1655@ "Employee SSN" ]
x167: 1628@ name[ name 1657@ "Project Number" ]
x168: 1630@ name[ name 1656@ "Hours Worked" ]
x169: 1635@ name[ name 1662@ "person" ]
x170: 1637@ name[ name 1663@ "addr" ]
x171: 1639@ name[ name 1664@ "phone_number" ]
x172: 1641@ name[ name 1665@ "idl_univ" ]
x173: 1643@ name[ name 1666@ "integer" ]
x174: 1645@ name[ name 1667@ "integer" ]
x175: 1647@ name[ name 1668@ "rational" ]
x176: 3012@ "Mathew"
x177: 3013@ "James"
x178: 1073@ "Rothlisberger"
x179: 3014@ "23 Feb 60"
x180: 3210@ person[ fname x176^ ; mname x177^ ; lname x178^ ; bdate x179^ ; ssn
122121212 ; spouse 3208@ "" ; sptr nil ]
x181: 3017@ "231 Bergen"
x182: 3018@ "Monterey"
x183: 3019@ "CA"
x184: 3020@ "93940"
x185: 3211@ addr[ street x181^ ; city x182^ ; state x183^ ; zip x184^ ]
x186: 3016@ "(408)375-1234"
x187: 3212@ phone_number[ number x186^ ]
x188: 3209@ cart1_result_tuple[ person x180^ ; address x185^ ; phone x187^ ; widget
nil ; essn 550926190 ; proj_num 2 ; hours 10.0 ]
x189: 3225@ person[ fname x176^ ; mname x177^ ; lname x178^ ; bdate x179^ ; ssn
122121212 ; spouse 3223@ "" ; sptr nil ]
x190: 3226@ addr[ street x181^ ; city x182^ ; state x183^ ; zip x184^ ]
x191: 3227@ phone_number[ number x186^ ]
x192: 3224@ cart1_result_tuple[ person x189^ ; address x190^ ; phone x191^ ; widget
nil ; essn 999999999 ; proj_num 1 ; hours 45.2 ]
x193: 3240@ person[ fname x176^ ; mname x177^ ; lname x178^ ; bdate x179^ ; ssn
122121212 ; spouse 3238@ "" ; sptr nil ]
x194: 3241@ addr[ street x181^ ; city x182^ ; state x183^ ; zip x184^ ]
x195: 3242@ phone_number[ number x186^ ]
x196: 3239@ cart1_result_tuple[ person x193^ ; address x194^ ; phone x195^ ; widget
nil ; essn 123456789 ; proj_num 2 ; hours 51.5 ]
x197: 3255@ person[ fname x176^ ; mname x177^ ; lname x178^ ; bdate x179^ ; ssn
122121212 ; spouse 3253@ "" ; sptr nil ]
x198: 3256@ addr[ street x181^ ; city x182^ ; state x183^ ; zip x184^ ]
x199: 3257@ phone_number[ number x186^ ]
x200: 3254@ cart1_result_tuple[ person x197^ ; address x198^ ; phone x199^ ; widget
nil ; essn 987654321 ; proj_num 30 ; hours 4.0 ]
x201: 3021@ "Leonard"
x202: 3022@ "H."
x203: 1079@ "Tharpe"
x204: 3023@ "12 Aug 58"
```

```
x205: 3270@ person[ fname x201^ ; mname x202^ ; lname x203^ ; bdate x204^ ; ssn
110121212 ; spouse 3268@ "" ; sptr nil ]
x206: 3026@ "432 Caldwell Drive"
x207: 3027@ "Monterey"
x208: 3028@ "CA"
x209: 3029@ "93940"
x210: 3271@ addr[ street x206^ ; city x207^ ; state x208^ ; zip x209^ ]
x211: 3025@ "(408)452-1234"
x212: 3272@ phone_number[ number x211^ ]
x213: 3269@ cart1_result_tuple[ person x205^ ; address x210^ ; phone x212^ ; widget
nil ; essn 550926190 ; proj_num 2 ; hours 10.0 ]
x214: 3285@ person[ fname x201^ ; mname x202^ ; lname x203^ ; bdate x204^ ; ssn
110121212 ; spouse 3283@ "" ; sptr nil ]
x215: 3286@ addr[ street x206^ ; city x207^ ; state x208^ ; zip x209^ ]
x216: 3287@ phone_number[ number x211^ ]
x217: 3284@ cart1_result_tuple[ person x214^ ; address x215^ ; phone x216^ ; widget
nil ; essn 999999999 ; proj_num 1 ; hours 45.2 ]
x218: 3300@ person[ fname x201^ ; mname x202^ ; lname x203^ ; bdate x204^ ; ssn
110121212 ; spouse 3298@ "" ; sptr nil ]
x219: 3301@ addr[ street x206^ ; city x207^ ; state x208^ ; zip x209^ ]
x220: 3302@ phone_number[ number x211^ ]
x221: 3299@ cart1_result_tuple[ person x218^ ; address x219^ ; phone x220^ ; widget
nil ; essn 123456789 ; proj_num 2 ; hours 51.5 ]
x222: 3315@ person[ fname x201^ ; mname x202^ ; lname x203^ ; bdate x204^ ; ssn
110121212 ; spouse 3313@ "" ; sptr nil ]
x223: 3316@ addr[ street x206^ ; city x207^ ; state x208^ ; zip x209^ ]
x224: 3317@ phone_number[ number x211^ ]
x225: 3314@ cart1_result_tuple[ person x222^ ; address x223^ ; phone x224^ ; widget
nil ; essn 987654321 ; proj_num 30 ; hours 4.0 ]
x226: 3030@ "Charles"
x227: 3031@ "L."
x228: 1085@ "Baumann"
x229: 3032@ "23 Jun 54"
x230: 3330@ person[ fname x226^ ; mname x227^ ; lname x228^ ; bdate x229^ ; ssn
220121212 ; spouse 3328@ "" ; sptr nil ]
x231: 3035@ "12345 General Lane"
x232: 3036@ "Ft Bragg"
x233: 3037@ "NC"
x234: 3038@ "16234"
x235: 3331@ addr[ street x231^ ; city x232^ ; state x233^ ; zip x234^ ]
x236: 3034@ "(122)324-9876"
x237: 3332@ phone_number[ number x236^ ]
x238: 3329@ cart1_result_tuple[ person x230^ ; address x235^ ; phone x237^ ; widget
nil ; essn 550926190 ; proj_num 2 ; hours 10.0 ]
x239: 3345@ person[ fname x226^ ; mname x227^ ; lname x228^ ; bdate x229^ ; ssn
220121212 ; spouse 3343@ "" ; sptr nil ]
x240: 3346@ addr[ street x231^ ; city x232^ ; state x233^ ; zip x234^ ]
x241: 3347@ phone_number[ number x236^ ]
x242: 3344@ cart1_result_tuple[ person x239^ ; address x240^ ; phone x241^ ; widget
nil ; essn 999999999 ; proj_num 1 ; hours 45.2 ]
x243: 3360@ person[ fname x226^ ; mname x227^ ; lname x228^ ; bdate x229^ ; ssn
220121212 ; spouse 3358@ "" ; sptr nil ]
x244: 3361@ addr[ street x231^ ; city x232^ ; state x233^ ; zip x234^ ]
x245: 3362@ phone_number[ number x236^ ]
x246: 3359@ cart1_result_tuple[ person x243^ ; address x244^ ; phone x245^ ; widget
nil ; essn 123456789 ; proj_num 2 ; hours 51.5 ]
x247: 3375@ person[ fname x226^ ; mname x227^ ; lname x228^ ; bdate x229^ ; ssn
220121212 ; spouse 3373@ "" ; sptr nil ]
x248: 3376@ addr[ street x231^ ; city x232^ ; state x233^ ; zip x234^ ]
x249: 3377@ phone_number[ number x236^ ]
x250: 3374@ cart1_result_tuple[ person x247^ ; address x248^ ; phone x249^ ; widget
nil ; essn 987654321 ; proj_num 30 ; hours 4.0 ]
x251: 1091@ "Ronald"
x252: 1092@ "L."
x253: 1093@ "Spear"
x254: 3039@ "29 Dec 62"
x255: 3390@ person[ fname x251^ ; mname x252^ ; lname x253^ ; bdate x254^ ; ssn
120926190 ; spouse 3388@ "" ; sptr nil ]
```

```
x256: 3042@ "397B Ricketts Road"
x257: 3043@ "Monterey"
x258: 3044@ "CA"
x259: 3045@ "93940"
x260: 3391@ addr[ street x256^ ; city x257^ ; state x258^ ; zip x259^ ]
x261: 3041@ "(408)375-8619"
x262: 3392@ phone_number[ number x261^ ]
x263: 3389@ cart1_result_tuple[ person x255^ ; address x260^ ; phone x262^ ; widget
nil ; essn 550926190 ; proj_num 2 ; hours 10.0 ]
x264: 3405@ person[ fname x251^ ; mname x252^ ; lname x253^ ; bdate x254^ ; ssn
120926190 ; spouse 3403@ "" ; sptr nil ]
x265: 3406@ addr[ street x256^ ; city x257^ ; state x258^ ; zip x259^ ]
x266: 3407@ phone_number[ number x261^ ]
x267: 3404@ cart1_result_tuple[ person x264^ ; address x265^ ; phone x266^ ; widget
nil ; essn 999999999 ; proj_num 1 ; hours 45.2 ]
x268: 3420@ person[ fname x251^ ; mname x252^ ; lname x253^ ; bdate x254^ ; ssn
120926190 ; spouse 3418@ "" ; sptr nil ]
x269: 3421@ addr[ street x256^ ; city x257^ ; state x258^ ; zip x259^ ]
x270: 3422@ phone_number[ number x261^ ]
x271: 3419@ cart1_result_tuple[ person x268^ ; address x269^ ; phone x270^ ; widget
nil ; essn 123456789 ; proj_num 2 ; hours 51.5 ]
x272: 3435@ person[ fname x251^ ; mname x252^ ; lname x253^ ; bdate x254^ ; ssn
120926190 ; spouse 3433@ "" ; sptr nil ]
x273: 3436@ addr[ street x256^ ; city x257^ ; state x258^ ; zip x259^ ]
x274: 3437@ phone_number[ number x261^ ]
x275: 3434@ cart1_result_tuple[ person x272^ ; address x273^ ; phone x274^ ; widget
nil ; essn 987654321 ; proj_num 30 ; hours 4.0 ]
x276: 1099@ "Jon"
x277: 1100@ "Lewis"
x278: 1101@ "Spear"
x279: 3046@ "16 Sep 58"
x280: 3450@ person[ fname x276^ ; mname x277^ ; lname x278^ ; bdate x279^ ; ssn
123456789 ; spouse 3448@ "" ; sptr nil ]
x281: 3048@ "3122 Apt B Sunset Strip"
x282: 1103@ "Redondo Beach"
x283: 1104@ "CA"
x284: 3049@ "99812"
x285: 3451@ addr[ street x281^ ; city x282^ ; state x283^ ; zip x284^ ]
x286: 3047@ "(301)322-2341"
x287: 3452@ phone_number[ number x286^ ]
x288: 3449@ cart1_result_tuple[ person x280^ ; address x285^ ; phone x287^ ; widget
nil ; essn 550926190 ; proj_num 2 ; hours 10.0 ]
x289: 3465@ person[ fname x276^ ; mname x277^ ; lname x278^ ; bdate x279^ ; ssn
123456789 ; spouse 3463@ "" ; sptr nil ]
x290: 3466@ addr[ street x281^ ; city x282^ ; state x283^ ; zip x284^ ]
x291: 3467@ phone_number[ number x286^ ]
x292: 3464@ cart1_result_tuple[ person x289^ ; address x290^ ; phone x291^ ; widget
nil ; essn 999999999 ; proj_num 1 ; hours 45.2 ]
x293: 3480@ person[ fname x276^ ; mname x277^ ; lname x278^ ; bdate x279^ ; ssn
123456789 ; spouse 3478@ "" ; sptr nil ]
x294: 3481@ addr[ street x281^ ; city x282^ ; state x283^ ; zip x284^ ]
x295: 3482@ phone_number[ number x286^ ]
x296: 3479@ cart1_result_tuple[ person x293^ ; address x294^ ; phone x295^ ; widget
nil ; essn 123456789 ; proj_num 2 ; hours 51.5 ]
x297: 3495@ person[ fname x276^ ; mname x277^ ; lname x278^ ; bdate x279^ ; ssn
123456789 ; spouse 3493@ "" ; sptr nil ]
x298: 3496@ addr[ street x281^ ; city x282^ ; state x283^ ; zip x284^ ]
x299: 3497@ phone_number[ number x286^ ]
x300: 3494@ cart1_result_tuple[ person x297^ ; address x298^ ; phone x299^ ; widget
nil ; essn 987654321 ; proj_num 30 ; hours 4.0 ]
x301: 3050@ "Jon"
x302: 3051@ "K."
x303: 1352@ "Walter"
x304: 3052@ "24 Dec 61"
x305: 3510@ person[ fname x301^ ; mname x302^ ; lname x303^ ; bdate x304^ ; ssn
991221234 ; spouse 3508@ "" ; sptr nil ]
x306: 3055@ "3321 City St."
x307: 3056@ "Marina"
```

```
x308: 3057@ "CA"
x309: 3058@ "93940"
x310: 3511@ addr[ street x306^ ; city x307^ ; state x308^ ; zip x309^ ]
x311: 3054@ "(408)122-4253"
x312: 3512@ phone_number[ number x311^ ]
x313: 3509@ cart1_result_tuple[ person x305^ ; address x310^ ; phone x312^ ; widget
nil ; essn 550926190 ; proj_num 2 ; hours 10.0 ]
x314: 3525@ person[ fname x301^ ; mname x302^ ; lname x303^ ; bdate x304^ ; ssn
991221234 ; spouse 3523@ "" ; sptr nil ]
x315: 3526@ addr[ street x306^ ; city x307^ ; state x308^ ; zip x309^ ]
x316: 3527@ phone_number[ number x311^ ]
x317: 3524@ cart1_result_tuple[ person x314^ ; address x315^ ; phone x316^ ; widget
nil ; essn 999999999 ; proj_num 1 ; hours 45.2 ]
x318: 3540@ person[ fname x301^ ; mname x302^ ; lname x303^ ; bdate x304^ ; ssn
991221234 ; spouse 3538@ "" ; sptr nil ]
x319: 3541@ addr[ street x306^ ; city x307^ ; state x308^ ; zip x309^ ]
x320: 3542@ phone_number[ number x311^ ]
x321: 3539@ cart1_result_tuple[ person x318^ ; address x319^ ; phone x320^ ; widget
nil ; essn 123456789 ; proj_num 2 ; hours 51.5 ]
x322: 3555@ person[ fname x301^ ; mname x302^ ; lname x303^ ; bdate x304^ ; ssn
991221234 ; spouse 3553@ "" ; sptr nil ]
x323: 3556@ addr[ street x306^ ; city x307^ ; state x308^ ; zip x309^ ]
x324: 3557@ phone_number[ number x311^ ]
x325: 3554@ cart1_result_tuple[ person x322^ ; address x323^ ; phone x324^ ; widget
nil ; essn 987654321 ; proj_num 30 ; hours 4.0 ]
x326: 1670@ person[ fname 1671@ "" ; mname 1672@ "" ; lname 1673@ "" ; bdate 1674@
"" ; ssn 0 ; spouse 1675@ "" ; sptr nil ]
x327: 1676@ addr[ street 1677@ "" ; city 1678@ "" ; state 1679@ "" ; zip 1680@ "" ]
x328: 1681@ phone_number[ number 1682@ "" ]
x329: 1669@ cart1_result_tuple[ person x326^ ; address x327^ ; phone x328^ ; widget
nil ; essn 0 ; proj_num 0 ; hours 0.0 ]
x330: 1605@ relation[ relation_name 1610@ "Cart Result1" ; attribute_names 1615@
< x162^ x163^ x164^ x165^ x166^ x167^ x168^ > ; attribute_types 1632@ < x169^ x170^
x171^ x172^ x173^ x174^ x175^ > ; tuples 3197@ < x188^ x192^ x196^ x200^ x213^
x217^ x221^ x225^ x238^ x242^ x246^ x250^ x263^ x267^ x271^ x275^ x288^ x292^ x296^
x300^ x313^ x317^ x321^ x325^ > ; tuple_type x329^ ; key 1611@ "ssn,essn,proj_num"
]
x331: 2793@ name[ name 2797@ "Hours Worked" ]
x332: 2795@ name[ name 2798@ "Employee SSN" ]
x333: 2800@ name[ name 2804@ "rational" ]
x334: 2802@ name[ name 2805@ "integer" ]
x335: 2809@ project1_result_tuple[ hours 20.4 ; essn 550926190 ]
x336: 2815@ project1_result_tuple[ hours 3.2 ; essn 123456789 ]
x337: 2818@ project1_result_tuple[ hours 67.25 ; essn 987654321 ]
x338: 2806@ project1_result_tuple[ hours 0.0 ; essn 0 ]
x339: 2784@ relation[ relation_name 2887@ "project 1" ; attribute_names 2792@ <
x331^ x332^ > ; attribute_types 2799@ < x333^ x334^ > ; tuples 2788@ < x335^ x336^
x337^ > ; tuple_type x338^ ; key 2791@ "essn" ]
x340: 2830@ proj_tuple[ essn 550926190 ; proj_num 2 ; hours 20.0 ]
x341: 2826@ proj_tuple[ essn 0 ; proj_num 0 ; hours 0.0 ]
x342: 2819@ relation[ relation_name 2825@ "comp obj" ; attribute_names 2821@ < >
; attribute_types 2822@ < > ; tuples 2823@ < x340^ > ; tuple_type x341^ ; key 2824@
"" ]
x343: 2838@ name[ name 2839@ "" ]
x344: 2842@ addr[ street 2843@ "" ; city 2844@ "" ; state 2845@ "" ; zip 2846@ "" ]
x345: 2848@ person[ fname 2849@ "" ; mname 2850@ "" ; lname 2851@ "" ; bdate 2852@
"" ; ssn 0 ; spouse 2853@ "" ; sptr nil ]
x346: 2854@ addr[ street 2855@ "" ; city 2856@ "" ; state 2857@ "" ; zip 2858@ "" ]
x347: 2859@ phone_number[ number 2860@ "" ]
x348: 2847@ emp_tuple[ person x345^ ; address x346^ ; phone x347^ ; widget nil ]
x349: 2861@ proj_tuple[ essn 0 ; proj_num 0 ; hours 0.0 ]
x350: 2840@ result_tuple[ values 2841@ < x344^ x348^ x349^ > ]
x351: 2862@ result_tuple[ values 2863@ < > ]
x352: 2832@ relation[ relation_name 2837@ "result test" ; attribute_names 2833@ <
x343^ > ; attribute_types 2835@ < > ; tuples 2836@ < x350^ x351^ > ; tuple_type
nil ; key 2831@ "" ]
x353: 1112@ "person -> ssn"
```

```
x354: 3563@ relation[ relation_name 3564@ "1TEMP_r1.r4" ; attribute_names 3565@ <
x1^ x2^ x3^ x4^ > ; attribute_types x39^ ; tuples 3566@ < x12^ x16^ x20^ x24^ x28^
x32^ x128^ x51^ > ; tuple_type x36^ ; key x353^ ]
x355: 1251@ database[ name 2864@ "Relational Address DB" ; relations 1253@ < x37^
x56^ x73^ x113^ x120^ x133^ x138^ x150^ x161^ x330^ x339^ x342^ x352^ x354^ > ]
```

APPENDIX E: MODIFIED R/OODBMS SCHEMA

```
*******************************************************************************
```

A Modified Relational/Object-Oriented Database Management System

```
*******************************************************************************

--============================================================================
--Description : This file contains the IDL schema for the implementation of
--              A Modified Relational/Object-Oriented Database Management
--              System. Of the five primitive relational algebra operations
--              (union, difference, project, select, and Cartesian product),
--              the project and Cartesian product operations have been modified
--              from the original R/OODBMS implemented in IDB.
--============================================================================


structure relational root database is

--*****************************************************************************

   database =>    name              : string,
                  relations         : seq of relation;

   database ->    new_relation(*);


   relation =>    relation_name     : string,
                  attribute_names   : seq of name,
                  attribute_types   : seq of name,
                  tuples            : seq of tuple,
                  tuple_type        : tuple,
                  key               : string;

   relation ->    new_tuple(*),
                  check_union_compatability(relation,relation) => boolean;

   relation ->    union(*),
                  projection(*),
                  difference(*),
                  Cartesian_product(*),
                  selection(*);

      name =>     name              : string;


   tuple ->       equal_to(tuple,tuple,integer)              => boolean,
                  less_than(tuple,tuple,integer)             => boolean,
                  greater_than(tuple,tuple,integer)          => boolean,

                  initialize_tuple(tuple)                    => tuple,
                  insert_fields(relation,relation)           => relation,

                  insert_fields_b(relation,relation,index_array) => relation,
                  -- the b extension indicates modified methods used in this
                  -- implementation.

                  insert_tuples(relation,relation,relation) => relation,

                  insert_tuples_b(relation,relation,relation)   => relation;
```

```
   tuple ::=      emp_tuple |
                  proj_tuple |
                  cart1_result_tuple |
                  project1_result_tuple |
                  result_tuple |
                  nil;
                  -- in this implementation, result_tuple is the only tuple
                  -- that needs to be defined for creating resultant relations
                  -- in both the project and Cartesian product operations. Thus,
                  -- cart1_result_tuple and project1_result_tuple are not needed
                  -- in this implementation. However, they have been left in so
                  -- the reader could more easily compare this schema with that
                  -- of the standard R/OODBMS schema.

--*****************************************************************************

   for database.new_relation use browser_visible;

   for relation.new_tuple use browser_visible;
   for relation.union use browser_visible;
   for relation.Cartesian_product use browser_visible;
   for relation.difference use browser_visible;
   for relation.projection use browser_visible;
   for relation.selection use browser_visible;


   for database.relations use linked;
   for relation.tuples use linked;

--*****************************************************************************

   for database.idl_key use bind(database_key);
   for database.idl_print use bind(database_print);
   for database.new_relation use bind(create_relation);

   for relation.idl_key use bind(relation_key);
   for relation.idl_print use bind(relation_print);
   for relation.new_tuple use bind(create_tuple);
   for relation.check_union_compatability use bind(ck_union_compatability);

   for relation.union use bind(union_op);
   for relation.Cartesian_product use bind(cart_prod_op);
   for relation.difference use bind(set_diff_op);
   for relation.projection use bind(project_op);
   for relation.selection use bind(select_op);

   for name.idl_key use bind(name_key);
   for name.idl_print use bind(name_print);

   for tuple.equal_to use bind(equal_to);
   for tuple.less_than use bind(less_than);
   for tuple.greater_than use bind(greater_than);

   for tuple.initialize_tuple use bind(initialize_tuple);
   for tuple.insert_fields use bind(insert_fields);
   for tuple.insert_tuples use bind(insert_tuples);

   for tuple.insert_fields_b use bind(insert_fields_b);
   for tuple.insert_tuples_b use bind(insert_tuples_b);


--*************************************emp_tuple*********************************

   emp_tuple =>         person   :  person,
                        address  :  addr,
```

```
                        phone     :  phone_number,
                        widget    :  idl_univ;

      person =>         fname     :  string,
                        mname     :  string,
                        lname     :  string,
                        bdate     :  string,
                        ssn       :  integer,
                        spouse    :  string,
                        sptr      :  person_nil;

         person_nil ::= person | nil;

      addr =>           street    :  string,
                        city      :  string,
                        state     :  string,
                        zip       :  string;

      phone_number =>   number    : string;

--*************************************************************************

   for emp_tuple.idl_key use bind(emp_tuple_key);
   for emp_tuple.idl_print use bind(emp_tuple_print);
   for emp_tuple.equal_to use bind(emp_equal_to);
   for emp_tuple.less_than use bind(emp_less_than);
   for emp_tuple.greater_than use bind(emp_greater_than);
   for emp_tuple.initialize_tuple use bind(initialize_emp_tuple);

   for emp_tuple.insert_fields_b use bind(insert_emp_fields_b);

   for person.idl_key use bind(person_key);
   for person.idl_print use bind(person_print);

   for addr.idl_key use bind(addr_key);
   for addr.idl_print use bind(addr_print);

   for phone_number.idl_print use bind(phone_number_print);

--*********************************proj_tuple*********************************

   proj_tuple =>           essn     : integer,
                           proj_num : integer,
                           hours    : rational;

--*************************************************************************

   for proj_tuple.idl_key use bind(proj_tuple_key);
   for proj_tuple.idl_print use bind(proj_tuple_print);
   for proj_tuple.equal_to use bind(proj_equal_to);
   for proj_tuple.less_than use bind(proj_less_than);
   for proj_tuple.greater_than use bind(proj_greater_than);
   for proj_tuple.initialize_tuple use bind(initialize_proj_tuple);

--*********************************cart1_result_tuple**************************

   cart1_result_tuple =>   person   :  person,
                           address  :  addr,
                           phone    :  phone_number,
                           widget   :  idl_univ,
                           essn     :  integer,
                           proj_num :  integer,
                           hours    :  rational;

--*************************************************************************

   for cart1_result_tuple.idl_key use bind(cart1_result_tuple_key);
```

```
   for cart1_result_tuple.idl_print use bind(cart1_result_tuple_print);
   for cart1_result_tuple.initialize_tuple use
                                          bind(initialize_cart1_resul_tuple);
   for cart1_result_tuple.insert_tuples use bind(insert_cart1_result_tuples);

--*****************************************************************************

   project1_result_tuple   => hours :  rational,
                              essn  :  integer;

--*****************************************************************************

   for project1_result_tuple.idl_key use bind(project1_result_tuple_key);
   for project1_result_tuple.idl_print use bind(project1_result_tuple_print);
   for project1_result_tuple.initialize_tuple use
                                       bind(initialize_project1_resul_tuple);
  for project1_result_tuple.insert_fields use bind(insert_project1_result_flds);

--*****************************************************************************

   result_tuple      => values  : seq of any;

--*****************************************************************************

   for result_tuple.values use linked;


--************************************Misc*************************************

   -- index_array is used by the project operation to hold a list of indexes
   -- to the attributes of a relation that are to be projected. It cannot be
   -- reached from the database root and therefore can never be stored in the
   -- database by accident. It is purely set up as a data structure to allow
   -- the indexes to be passed as a parameter in the modified function
   -- project_op.
   index_array => indexes : seq of index;

         index => i : integer;

   for index_array.indexes use linked;


--*****************************************************************************

end

--*****************************************************************************

process relationalp is

   relational ::= relationala:access;

end
```

APPENDIX F: MODIFIED PROJECT

```
*****************************************************************************
                          Project_parse_action
*****************************************************************************

static void Project_parse_action(query)
     char* query;
{
  char *R1_ptr,*char_ptr,*R2_ptr;
  integer size,i;
  boolean done = false,delimeter1 = false;

  char_ptr = query;

  /* allocate room for parse of the project op parameters
     R1 will hold the relation being operated on,
     Attr_list the list of attr to be projected, and
     R2 the resultant relation  */
  size = strlen(query);
  R1 = (char*)calloc((size+1),sizeof(char)); /* R1 is global */
  Attr_list = (char*)calloc((size+1),sizeof(char));

  /* set pointers to move along R1 and R2 as characters are copied in one
     at a time. */
  R1_ptr = R1;

  /* do the parse */
  char_ptr = query;

  /* note: if size gets decremented all the way to zero, then there is a
     problem with the query because the delimeter ' = ' or ' project '
     could not be found */
  while (!done && size > 0)
    {
      if (*char_ptr != ' ') /* if not a space copy the char into R1 */
        {
          *R1_ptr=*char_ptr;/*parameter relation*/
          ++char_ptr;
          ++R1_ptr;
          --size;
        }
      else /* we may have hit a delimeter */
        {
          /* check to see if next char is a " p " - which is
             part of the delimeter " project "between the last
             two parameters */

          if (strncmp(char_ptr," project ",9) == 0)
            /* then it is project */
            {
              for (i = 0; i < 9;++i) /* jump past the delimeter */
                {
                  ++char_ptr;
                  --size;
                }
              strcpy(Attr_list,char_ptr);
              /* copy list of attributes into Attr_list and now parse the
                 list of attributes */
              done=true;
```

```
            }
          else /* the space is part of the first parameter, so put in R1 */
            {
              /* space is part of first relation name so keep it */
              *R1_ptr=*char_ptr;
              ++char_ptr;
              ++R1_ptr;
              --size;
            }
        }
    }

  if (size != 0) /* size only = 0 if union was not found in the query */
    NULL;
  else
    idl_raise(IDL_ERROR,"There is an error in your query! Try Again.");
}
```

```
*****************************************************************************
```

report_project_error

```
*****************************************************************************

void report_project_error(found1,found2,attr_found)
     boolean found1, found2, attr_found;
{
  if (!found1 && !found2)
    {
      idl_raise(IDL_ERROR,
                "Neither of the two relations are in this database!");
    }
  else
    {
      if (!attr_found)
        {
          idl_raise(IDL_ERROR,
                  "All of the attributes in the attribute list \nare not in R1!");
        }
      if (!found1)
        {
          idl_raise(IDL_ERROR,
                    "R1 is not in this database!");
        }
      if (!found2)
        {
          idl_raise(IDL_ERROR,
                    "R2 is not in this database!");
        }

      if (found1 && found2)
        {
          idl_raise(IDL_ERROR,
                    "A SERIOUS ERROR HAS OCCURED !!!!!! Regroup. Try Again.");
        }
    }
}
```

project_op

```
idl_routine void project_op(relation)
     relational_relation relation;
{
  relational_relation ptr_R1,ptr_Attr_list,ptr_result_rel,temp_relation;
  relational_database database;
  relational_index_array array_index; /*beta project*/
  relational_index attr_index; /*beta project*/
  idl_trans_mode tmode;
  idl_univ root;
  string parameter1,attr_string;  /* references to the parameter
                                     R1,and  attr list
                                     relations  respectively */
  idl_transaction tr;
  boolean found1,found2,done,attr_found;
  boolean is_writable = false,duplicate = true;
  char *attr_ptr,*delimeter = ",";  /* delimeter between elements
                                       in attribute list */

  string *attr_list[100];
  integer i=1,count,size,index,index_array[100],ii=0;
  /* index array of size 100 allows a relation to have 100 attributes */
  idl_linked_elem(relational_tuple) result_tuple;

  tr = idl_get_trans(relation);
  tmode = idl_trans_mode_default;
  root = idl_trans_get_root(tr);
  database = idl_to(relational_database,root);
  found1 = false;
  found2 = true;/* not needed for beta version, thus changed to true*/
  done = false;
  is_writable = (idl_trans_write_count(tr) > 0);

  array_index = idl_new(tr,relational_index_array);/*creates the index array
                                                     to be used in the beta
                                                     version of this op*/
  array_index->indexes = idl_empty_linked(tr,relational_index);


  brw_input("Project Query",
            "Please input the Project query (R1 project Attr_list): ",
            0L,0L,0L,false,
            Project_parse_action);


  /* copy the C strings R1, R2 and Attr_list into IDL strings */
  parameter1 = idl_copy_string(tr,R1); /*relation being operated on*/
  attr_string = idl_copy_string(tr,Attr_list);

  /* parse tokens in attribute string */
  if ((attr_ptr = strtok(attr_string,delimeter)) == NULL)
    {
      /* error, no token */

      idl_raise(IDL_ERROR,
             "You did not list any attribute/field names in\nyour project query!
Try again, meathead!");
    }
  else
    {
      attr_list[0] = idl_new_string(tr,80);
      attr_list[0] = idl_copy_string(tr,attr_ptr);
    }
```

```
while ((attr_ptr = strtok(NULL,delimeter)) != NULL)
  {
    attr_list[i] = idl_new_string(tr,80);
    attr_list[i] = idl_copy_string(tr,attr_ptr);
    i++;
  }

/* search the database for the relation R1 */
idl_linked_for (relational_relation,database->relations,rel)
  {
    if (strcmp (rel->relation_name,parameter1) == 0)
      /* found relation 1 */
      {
        ptr_R1 = rel; /* point at relation 1 */
        found1 = true;
      }
  } idl_end_for

count = i; /* count is the number of tokens - 1 */

/* check each attr name in the attr list of the project operation to
   ensure that the field exists in the relation R1 */

for(i=0;i<count;++i)
  {
    attr_found = false;
    ii=0;

    idl_array_for(relational_name,ptr_R1->attribute_names,aname)
      {
        ii++; /* position in attribute list */
        if (strcmp (aname->name,attr_list[i]) == 0) /* attr name in attr
                                                       list is a field
                                                       of R1 */
          {
            attr_found = true;
            attr_index = idl_new(tr,relational_index);
            attr_index->i = ii;
            idl_insert_back(relational_index,
                            array_index->indexes,attr_index);
            break;
          }
      } idl_end_for

    if (!attr_found)
      {
        break; /* an attr in the project attr list is not
                  in the relation R1. Thus, the operation
                  cannot be performed */
      }
  }

if (found1 && attr_found)
  {
    /* everything is ok, perform projection operation on relation R1.
       Note, in this implementation, the resultant relation doesn't
       already exists in the database. */

    ptr_result_rel = init_proj_result_rel(ptr_R1,array_index);

   ptr_result_rel=idl_vop(ptr_R1->tuple_type,relational_tuple,insert_fields_b,
                          (ptr_R1,ptr_result_rel,array_index));

    idl_insert_back(relational_relation,database->relations,ptr_result_rel);
  }
else
  {
```

```
        report_project_error(found1,found2,attr_found);
    }
}
```

```
*****************************************************************************
```

insert_fields_b

```
*****************************************************************************

idl_routine relational_relation insert_fields_b(rel,result_rel,index_array)
     relational_relation rel, result_rel;
     relational_index_array index_array;
{
  return result_rel;
}
```

insert_emp_fields_b

```
idl_routine relational_relation insert_emp_fields_b(rel,result_rel,index_array)
     relational_relation rel, result_rel;
     relational_index_array index_array;
{
  idl_transaction tr = idl_get_trans(rel);
  relational_result_tuple new_tuple;

  result_rel->tuples = idl_empty_linked(tr,relational_tuple);

  idl_linked_for (relational_tuple,rel->tuples,rel_tuple)
    {
      /* iterate through each tuple and for each tuple iterate through
         the index_array and use a case statement to reference objects for
         fields to be entered into the result relation */

      new_tuple = idl_new(tr,relational_result_tuple);
      new_tuple->values = idl_empty_linked(tr,relational_any);

      idl_linked_for (relational_index,index_array->indexes,index)
        {
          switch (index->i)
            (
            case 1:
            idl_insert_back(relational_any,new_tuple->values,rel_tuple->person);
              break;
            case 2:
           idl_insert_back(relational_any,new_tuple->values,rel_tuple->address);
              break;
            case 3:
             idl_insert_back(relational_any,new_tuple->values,rel_tuple->phone);
              break;
            case 4:
              /* widget */
              break;
            default:
              idl_raise(IDL_ERROR,
                        "There is a problem in the employee insert field beta
function!");
              break;
            )
        } idl_end_for

      idl_insert_back(relational_tuple,result_rel->tuples,new_tuple);

    ) idl_end_for

  return result_rel;
)
```

```
*****************************************************************************
```

init_proj_result_rel

```
*****************************************************************************

relational_relation init_proj_result_rel(ptr_R1,index_array)
     relational_relation ptr_R1;
     relational_index_array index_array;
{
  relational_relation result_relation;
  static integer result_rel_num = 0;
  idl_transaction tr;
  string empty,result_rel_name;
  char result_name[80];
  integer degree = 0,i;

  tr = idl_get_trans(ptr_R1);

  i=0;
  idl_linked_for (relational_index,index_array->indexes,index)
    {
      i++;
    }idl_end_for

  degree = i;

  result_relation = idl_new(tr,relational_relation); /* must still assign
                                                        legal values */

  /* set up a unique name for resultant relation */
  sprintf(result_name,"%1dRESULT_%c%c",
          ++result_rel_num,
          ptr_R1->relation_name[0],
          ptr_R1->relation_name[1]);
  result_rel_name = idl_copy_string(tr,result_name);

  result_relation->relation_name = result_rel_name;
  result_relation->attribute_names = idl_new_array(tr,relational_name,degree);
  result_relation->attribute_types = idl_new_array(tr,relational_name,degree);

  /* assign default values for attribute names to be the same as those in
     R1 relation */
  i=0;
  idl_linked_for (relational_index,index_array->indexes,index)
    {
      result_relation->attribute_names[i]=ptr_R1->attribute_names[(index->i)-1];
      result_relation->attribute_types[i]=ptr_R1->attribute_types[(index->i)-1];
      i++;
    }idl_end_for

  result_relation->tuples = idl_empty_linked(tr,relational_tuple);

  /* assign a default tuple type that is the same as the first relations */
  result_relation->tuple_type = idl_to(relational_tuple,
                                       idl_new(tr,relational_result_tuple));

  /* default key is the key of relation R1 */
  result_relation->key = ptr_R1->key;

  return result_relation;
}
```

APPENDIX G: MODIFIED CARTESIAN PRODUCT

```
*****************************************************************************
                        Cartesian_parse_action
*****************************************************************************

static void Cartesian_parse_action(query)
     char* query;
{
  char *R1_ptr,*char_ptr;
  integer size,i;
  boolean done = false,delimeter1 = false;

  char_ptr = query;

  /* allocate room for parse of the Cartesian product op parameters
     R1 will hold the first parameter and R2 the second */
  size = strlen(query);
  R1 = (char*)calloc((size+1),sizeof(char));
  R2 = (char*)calloc((size+1),sizeof(char));

  /* set pointer to move along R1 as characters are copied in one
     at a time. */
  R1_ptr = R1;

  /* do the parse */
  char_ptr = query;

  /* note: if size gets decremented all the way to zero, then there is a
     problem with the query because the delimeter ' X ' could not be
     found */
  while (!done && size > 0)
    {
      if (*char_ptr != ' ') /* if not a space copy the char into R3 */
        {
          *R1_ptr=*char_ptr;
          ++char_ptr;
          ++R1_ptr;
          --size;
        }
      else /* we may have hit the a delimeter */
        {
          /* check to see if next char is a " X " - which separates the
             two operands of the operation */

          if (strncmp(char_ptr," X ",3) == 0) /* then it is X sentinal */
            {
              for (i = 0; i < 3;++i) /* jump past the delimeter */
                {
                  ++char_ptr;
                  --size;
                }
              strcpy(R2,char_ptr); /* copy second parameter into R2 */
              done=true;
            }
          else /* the space is part of the first parameter, so put in R1 */
            {
              /* space is part of first relation name so keep it */
              *R1_ptr=*char_ptr;
              ++char_ptr;
```

```
                ++R1_ptr;
                --size;
            }
        }
    }
}
```

```
*****************************************************************************
```

report_Cart_product_error

```
*****************************************************************************

void report_Cart_product_error(found1,found2,found3)
     boolean found1,found2,found3;
{
  if (!found1 && !found2)
    {
      idl_raise(IDL_ERROR,
                "Neither of the two relations are in this database!");
    }
  else
    {
      if (!found1)
        {
          idl_raise(IDL_ERROR,
                    "R1 is not in this database!");
        }
      if (!found2)
        {
          idl_raise(IDL_ERROR,
                    "R2 is not in this database!");
        }
      if (!found3)
        {
          idl_raise(IDL_ERROR,
                    "R3 is not in this database!");
        }
      if (found1 && found2 && found3)
        {
          idl_raise(IDL_ERROR,
                    "A SERIOUS ERROR HAS OCCURED !!!!!! Regroup. Try Again.");
        }
    }
}
```

```
*****************************************************************************
```

cart_prod_op

```
*****************************************************************************

idl_routine void cart_prod_op(relation)
     relational_relation relation;
{
  relational_relation ptr_R1,ptr_R2,resutl_rel,temp_relation;
  relational_database database;
  idl_trans_mode tmode;
  idl_univ root;
  string parameter1,parameter2,result_rel;  /* references to the parameters
                                               R1, R2 and R3 respectively */
  idl_transaction tr;
  boolean found1,found2,found3;
  boolean is_writable = false,duplicate = true;

  tr = idl_get_trans(relation);
  tmode = idl_trans_mode_default;
  root = idl_trans_get_root(tr);
  database = idl_to(relational_database,root);
  found1 = false;
  found2 = false;
  found3 = true; /* beta version doesn't use a predefined result rel */
  is_writable = (idl_trans_write_count(tr) > 0);

  brw_input("Cartesion Product Query",
            "Please input the Cartesian product query (R1 X R2): ",
            0L,0L,0L,false,
            Cartesian_parse_action);

  /* copy the C strings R1 and R2 into IDL strings */
  parameter1 = idl_copy_string(tr,R1);
  parameter2 = idl_copy_string(tr,R2);

  /* don't do anything if the resultant relation is one of the two operands.
     However, the resultant relation can be one that exists in the data.
     In this case, the specified resultant relation will be over written. */

  /* search the database for the two relations: R1, and R2 */
  idl_linked_for (relational_relation,database->relations,rel)
    {
      if (strcmp (rel->relation_name,parameter1) == 0)
        /* found relation 1 */
        {
          ptr_R1 = rel; /* point at relation 1 */
          found1 = true;
        }

      if (strcmp (rel->relation_name,parameter2) == 0)
        /* found relation 2 */
        {
          ptr_R2 = rel;
          found2 = true;
        }

    } idl_end_for


  if (found1 && found2)
    {
      /* perform concatenation of tuples for Cartesian product.
         Note, in this implementation, the resultant relation does not
         already exists in the database. */

      result_rel = init_Cart_result_rel(ptr_R1,ptr_R2);
```

```
      result_rel=idl_vop(result_rel->tuple_type,relational_tuple,insert_tuples_b,
                         (ptr_R1,ptr_R2,result_rel));

       idl_insert_back(relational_relation,database->relations,result_rel);
     }
   else
     {
       report_Cart_product_error(found1,found2,found3);
     }
}
```

****

init_Cart_result_rel

****

```
relational_relation init_Cart_result_rel(ptr_R1,ptr_R2)
     relational_relation ptr_R1,ptr_R2;
(
  relational_relation result_relation;
  static integer result_rel_num = 0;
  idl_transaction tr;
  string result_rel_name;
  char result_name[80];
  integer degree1 = 0,degree2 = 0,result_degree,i;

  tr = idl_get_trans(ptr_R1);

  degree1 = idl_array_size(ptr_R1->attribute_names);
  degree2 = idl_array_size(ptr_R2->attribute_names);
  result_degree = degree1 + degree2;

  result_relation = idl_new(tr,relational_relation); /* must still assign
                                                        legal values */

  /* set up a unique name for resultant relation */
  sprintf(result_name,"%1dRESULT_%c%c.%c%c",
          ++result_rel_num,
          ptr_R1->relation_name[0],
          ptr_R1->relation_name[1],
          ptr_R2->relation_name[0],
          ptr_R2->relation_name[1]);
  result_rel_name = idl_copy_string(tr,result_name);

  result_relation->relation_name = result_rel_name;
  result_relation->attribute_names = idl_new_array(tr,relational_name,
                                                   result_degree);
  result_relation->attribute_types = idl_new_array(tr,relational_name,
                                                   result_degree);

  /* assign default values for attribute names to be the same as those in
     R1 and R2  */

  for ( i=0; i<degree1; ++i)
    {
      result_relation->attribute_names[i] = ptr_R1->attribute_names[i];
      result_relation->attribute_types[i] = ptr_R1->attribute_types[i];
    }

  for ( i=0; i<degree2; ++i)
    {
      result_relation->attribute_names[degree1+i] = ptr_R2->attribute_names[i];
      result_relation->attribute_types[degree1+i] = ptr_R2->attribute_types[i];
    }

  result_relation->tuples = idl_empty_linked(tr,relational_tuple);

  /* assign a default tuple type result tuple */
  result_relation->tuple_type = idl_to(relational_tuple,
                                       idl_new(tr,relational_result_tuple));

  /* default key is the key of relation R1 */
  result_relation->key = ptr_R1->key;

  return result_relation;
)
```

insert_tuples_b

```
static relational_relation insert_tuples_b(r1,r2,result_rel)
     relational_relation r1,r2,result_rel;
{
  relational_result_tuple new_tuple;
  relational_tuple rel1,rel2;
  idl_transaction tr = idl_get_trans(r1);

  /* get rid of any tuples that may be in the resultant relation structure
     prior to insert the new result */

  result_rel->tuples = idl_empty_linked(tr,relational_tuple);

  idl_linked_for (relational_tuple,r1->tuples,r1_tuple)
    {
      idl_linked_for (relational_tuple,r2->tuples,r2_tuple)
        {
          /* send message to get a new tuple created with valid default
             values. */

          new_tuple = idl_new(tr,relational_result_tuple);
          new_tuple->values = idl_empty_linked(tr,relational_any);

          idl_insert_back(relational_any,new_tuple->values,r1_tuple);
          idl_insert_back(relational_any,new_tuple->values,r2_tuple);

          idl_insert_back(relational_tuple,result_rel->tuples,new_tuple);

        } idl_end_for
    } idl_end_for

  return result_rel;
}
```

LIST OF REFERENCES

[AHS91] Andrews, T., Harris, C., and Sinkel, K., "ONTOS: A Persistent Database for C++," in Gupta, R. and Horowitz, E. (Eds.), *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, Prentice Hall, Inc., Englewood Cliffs, NJ, pp. 387-406, 1991.

[BK90] Berri, C. and Kornatzky, Y., "Algebraic Optimization of Object-Oriented Query Languages," *Lecture Notes in Computer Science*, v. 470, S. Abiteboul and P. C. Kanellakis (Eds.), International Conference on Database3 Theory (ICDT) '90, Proceedings, pp. 72-88, Dec 1990.

[BM91] Bertino, E. and Martino, L., "Object-Oriented Database Management Systems: Concepts and Issues," *Computer*, v. 24, no. 4, pp. 33-47, Apr 1991.

[BMO89] Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E. H., Williams, M., "The GemStone Data Management System," in Kim, W. and Lochavsky, F. H. (Eds.), *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publishing Company, Inc., Reading, MA, pp. 283-308, 1989.

[BOS91] Butterworth, P., Otis, A. and Stein, J., "The Gemstone Object Database Management System," *Communications of the ACM*, v. 34, no. 10, pp. 64-77, Oct 1991.

[Cl91] Clark, G. J., *DFQL: A Graphical Dataflow Query Language*, Master's Thesis, Naval Postgraduate School, Monterey, CA, Sep 1991.

[CY90] Coad, P. and Yourdon, E., *Object-Oriented Analysis*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

[Co70] Codd, E. F., "A Relational Model for Large Shared Data Banks," *Communications of the ACM*, v. 13, no.6, pp. 377-387, Jun 1970.

[CD90] Interview between E. F. Codd and DBMS, "Relational philosopher: the creator of the relational model talks about his never-ending crusade," DBMS, v. 3, no. 13, pp. 34-42, Dec 1990.

[Da84] Date, C. J., *A Guide to DB2*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1984.

[Da90] Date, C. J., *An Introduction to Database Systems*, Fifth Edition, Volume 1, Addison-Wesley Publishing Company, Readin, MA, 1990.

[Ed91] Edelstein, H., "Relational vs. Object-Oriented," DBMS, v. 4, no. 12, pp. 68-74, Nov 1991.

[EN89] Elmasri, R. and Navathe, S. B., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1989.

[Fi92] Filippi, S. C., *Implementing Relational Operations in an Object-Oriented Database*, Master's Thesis, Naval Postgraduate School, Monterey, CA, Mar 1992.

[GH91a] Gupta, R. and Horowitz, E. (Eds.), *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1991.

[GH91b] Gupta, R. and Horowitz, E., "A Guide to the OODB Landscape", in Gupta, R. and Horowitz, E. (Eds.), *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, Prentice Hall, Inc., Englewood Cliffs, NJ, pp. 1-11, 1991.

[HO87] Halbert, D. C. and O'Brien, P. D., "Using Types and Inheritance in Object-Oriented Programming," *IEEE Software*, v. 4, no. 5, pp. 71-79, Sep 1987.

[HW91] Horowitz, E. and Wan, Q., "An Overview of Existing Object-Oriented Database Systems," in Gupta, R. and Horowitz, E. (Eds.), *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, Prentice Hall, Inc., Englewood Cliffs, NJ, pp. 101-116, 1991.

[Hs91] Hsiao, D. K., "The Object-Oriented Database Management - A Tutorial on its Fundamentals", Naval Postgraduate School, Monterey, CA, Aug 1991 (draft).

[In89] "Instances," *Release 1.0*, v. 89, no. 9, pp. 14-25, Sep 1989.

[KR78] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Engelwood Cliffs, NJ, 1978.

[Kh91] Khoshafian, S., "Modeling with object-oriented databases", *AI Expert*, v. 6, no. 10, pp. 26-34, Oct 1991.

[Ki91] Kim, H., "Algorithmic and Computational Aspects of OODB Schema Design," in Gupta, R. and Horowitz, E. (Eds.), Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, Prentice Hall, Inc., Englewood Cliffs, NJ, pp. 26-61, 1991.

[Ki90] Kim, W., "Research Directions in Object-Oriented Database Systems," in *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Nashville, Tennessee, pp. 1-15, Apr 1990.

[KL89] Kim, W. and Lochavsky, F. H. (Eds.), *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.

[Kin89] King, R., "My Cat is Object-Oriented", in Kim, W. and Lochavsky, F. H. (Eds.), *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publishing Company, Inc., Reading, MA, pp. 23-30, 1989.

[KS86] Korth, H. F. and Silberschatz, A., *Database System Concepts*, McGraw-Hill, Inc., New York, NY, 1986.

[Mi88] Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages," *JOOP*, v. 1, no. 1, pp. 12-34, Apr/May 1988.

[Mc91] McLeod, D., "A Perspective on Object-Oriented and Semantic Database Models and Systems," in Gupta, R. and Horowitz, E. (Eds.), *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, Prentice Hall, Inc., Englewood Cliffs, NJ, pp. 12-25, 1991.

[Me90] Meyer, P., "Are Object-Oriented Data Bases Ready For Business?," *Mainframe Update Magazine*, pp. 14-19, Autumn 1990.

[Mo89] Moon, D. A., "The COMMON LISP Object-Oriented Programming Language Standard", in Kim, W. and Lochavsky, F. H. (Eds.), *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley Publishing Company, Inc., Reading, MA, pp. 49-77, 1989.

[NMO90] Nelson, M. L., Moshell, J. M., and Orooji, A., "A Relational Object-Oriented Management System," *IEEE 1990 International Pheonix Conference on Computers and Communications (IPPCCC'90)*, Scottsdale, AZ, pp. 319-323, Mar 1990.

[Ne90a] Nelson, M. L., *Object-Oriented Database Management Systems*, Naval Postgraduate School, Monterey, CA, Report No NPS52-90-025, May 1990.

[Ne88] Nelson, M. L., *A Relational Object-Oriented Management System and an Encapsulated Object-Oriented Programming System.* Doctoral Dissertation, University of Central Florida, Orlando, FL, Dec 1988.

[Ne90b] Nelson, M. L., *An Introduction To Object-Oriented Programming*, Naval Postgraduate School, Monterey, CA, Report No NPS52-90-024, Apr 1990.

[Ne91] Nelson, M. L., "An Object-Oriented Tower of Babel", *OOPS Messenger*, v. 2, no. 3, pp. 3-11, Jul 1991.

[NMSW83] Nestor, J. R., Mishra, B., Scherlis, W. L. and Wulf, W. A., *Extensions to Attribute Grammars*, Tartan Laboratories Incorporated, Pittsburgh, PA, Technical Report TL 83-36, Apr 1983.

[New86] Newcomer, J. M., "IDL: Past Experience and New Ideas", *Lecture Notes in Computer Science*, Vol. 244, Conradi, R., Didriksen, T. M., and Wanvik, D. H. (Eds.), *Advanced Programming Environments, Proceedings of an International Workshop*, Trondheim, Norway, pp. 257-289, Jun 1986.

[Ni89] Nierstrasz, O., "A Survey of Object-Oriented Concepts", in Kim, W. and Lochavsky, F. H. (Eds.), Object-Oriented Concepts, Databases, and Applications, Addison-Wesley Publishing Company, Inc., Reading, MA, pp. 3-21, 1989.

[OV91] Ozsu, M. T. and Valduriez, P., *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[PN91b] de Paula, E. G. and Nelson, M. L., *An Object-Oriented Design Methodology*, Naval Postgraduate School, Monterey, CA, Report No NPSCS-91-007, Jan 1991.

[Pe91a] Persistent Data Systems, Inc., *IDB C Programmer's Manual, IDB Version 1.0*, Jan 1991.

[Pe91b] Persistent Data Systems, Inc., *IDB Release Notes, IDB Version 1.1*, Nov 1991.

[Pe91c] Persistent Data Systems, Inc., *IDB Tutorial, IDB Version 1.1*, Oct 1991.

[Pe91d] Persistent Data Systems, Inc., *IDB User's Manual, IDB Version 1.0*, Jan 1991.

[RK] Rhein, J. and Kemnitz, G. (Eds.) , *The POSTGRES User Manual*, EECS Department, University of California, Berkeley.

[Sc91] Schwartz, K. D., "Geode tools build object-oriented systems," Government Computer News, v. 10, no. 23, p. 49, 11 Nov 1991.

[SSU91] Silberschatz, A., Stonebraker, M. and Ullman, J., "Database Systems: Achievements and Opportunities," *Communications of the ACM*, v. 34, no. 10, pp. 110-120, Oct 1991.

[SB86] Stefik, M. and Bobrow, D. G., "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, v. 6, no. 4, pp. 40-62, Winter 1986.

[SK91] Stonebraker, M. and Kemnitz, G., "The POSTGRES Next Generation Database Management System," *Communications of the ACM*, v. 34, no. 10, pp. 78-92, Oct 1991.

[St88] Stonebraker, M., "Future Trends in Data Base Systems," *1988 IEEE Data Engineering Conference, Proceedings*, Los Angeles, CA, pp. 1-21, Feb 1988.

[St91a] Strehlo, K., "OODBMS pays off. (interview with Mike DeSanti)," *DBMS*, v. 4, pp. 48-54, Nov 1991.

[St91b] Strehlo, K., "The world according to Stonebraker: from Ingres to Postgres and the next generation of database management systems. (interview with Ingres developer and Ingres Corp. cofounder Michael Stonebraker)," *DBMS*, v. 4, no. 10, pp. 42-46, Sep 1991.

[St91c] Strehlo, K., "The OODBMS cutting edge," *DBMS*, v. 4, pp. 8-11, Nov 1991.

[US90] Unland, R. and Schlageter, G., "Object-Oriented Database Systems: Concepts and Perspectives," *Lecture Notes in Computer Science*, v. 466, A. Blaser (Ed.), Database Systems of the 90s, Proceedings, pp. 154-191, Nov 1990.

[We87] Wegner, P., "Dimensions of Object-Based Language Design", *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87) Conference Proceedings*, Oct 1987, Orlando, FL; special issue of *SIGPLAN Notices*, v. 22, no. 12, pp. 168-182, Dec 1987.

BIBLIOGRAPHY

[Cop92] Coplien, J. O., *Advanced C++ Programming Styles and Idioms*, Addison-Wesley Publishing Company, Readin, MA, 1992.

[Gh90] Ghelli, G., "A class abstraction for a hierarchical type system," *Lecture Notes in Computer Science*, Vol. 470, S. Abiteboul and P. C. Kanellakis (Eds.), *International Conference on Database3 Theory (ICDT) '90, Proceedings*, pp. 56-71, Dec 1990.

[LKM90] Lockemann, P. C., Kemper, A., and Moerkotte, G., "Future Database Technology: Driving Forces and Directions," *Lecture Notes in Computer Science*, v. 466, A. Blaser (Ed.), Database Systems of the 90s, Proceedings, pp. 15-33, Nov 1990.

[NWL81] Nestor, J. R., Wulf, W. A., and Lamb, D. A., *IDL - Interface Description Language* - Formal Description, Department of Computer Science, Carnegie-Mellon University, Technical Report, Aug 1981.

[Nes86] Nestor, J. R., "Toward a Persistent Object Base", *Lecture Notes in Computer Science*, Vol. 244, Conradi, R., Didriksen, T. M., and Wanvik, D. H. (Eds.), Advanced Programming Environments, Proceedings of an International Workshop, Trondheim, Norway, pp. 372-394, Jun 1986.

[PN91a] de Paula, E. G. and Nelson, M. L., "Designing a Class Hierarchy," *Technology of Object-Oriented Languages and Systems 5 (TOOLS 5)*, Santa Barbara, CA, pp. 203-218, Jul 1991.

[SS90] Scholl, M. H. and Schek, H., "A Relational Object Model", *Lecture Notes in Computer Science*, Vol. 470, S. Abiteboul and P. C. Kanellakis (Eds.), International Conference on Database3 Theory (ICDT) '90, Proceedings, pp. 89-105, Dec 1990.

[ZM90] Zdonik, S. B. and Maier, D. (Eds), *Readings in Object-Oriented Database Systems*, Morgan Kaufamann Publishers, Inc., San Mateo, CA, 1990.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Computer Science Dept. Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
4.	MAJ M. L. Nelson, USAF, Code CS/Ne Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
5.	C. Thomas Wu, Code CS/Wq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5002	2
6.	Dr. A. Orooji Computer Science Department University of Central Florida Orlando, FL 32816	1
7.	CPT Ronald L. Spear, USA 161 Oakdale Drive Zelienople, PA 16063	4
8.	John Nestor and Ellen Borison Persistent Data Systems, Inc. 75 West Chapel Ridge Road Pittsburgh, PA 15238	1
9.	LTC Robert Butler, USA 246 South Oakwood Drive Novato, CA 94949	1

10. CPT (P) Matthew James Rothlisberger 1
P.O. Box 3362
Fort Leavenworth, KS 66027

845-216